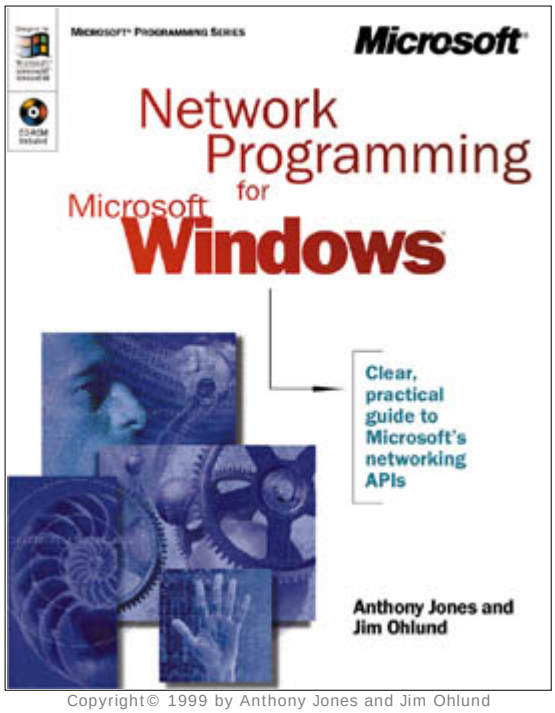

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Jones, Anthony, 1973-
Network Programming for Microsoft Windows / Anthony Jones, Jim Ohlund.
p. cm.
Includes index.
ISBN 0-7356-0560-2
1. Internet programming. 2. Microsoft Windows (Computer file)
I. Ohlund, Jim, 1966-. II. Title.
QA76.625.J65 1999
005.7'1276--dc21 99-34581

Printed and bound in the United States of America.

123456789 QMQM 4 3 2 1 0 9

Distributed in Canada by Penguin Books Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at mspress.microsoft.com.

Acquisitions Editor: Ben Ryan

Project Editor: Rebecca McKay

Technical Editor: Donnie Cameron

For my loving wife, Genevieve,
thanks for your patience and understanding.
–A.J.

For Samantha
–J.O.

Acknowledgments

We would like to thank the following individuals for their generous contributions to the making of this book. A very special thanks goes out to Wei Hua for developing all of the Visual Basic examples on our companion CD-ROM. Wei also helped us understand I/O completion ports and transport service providers in the Winsock 2 SPI. Thanks to Barry Butterklee for helping us understand I/O completion ports. Barry also reviewed the QOS chapter to ensure its accuracy. Amol Deshpande provided valuable information on IFS handles in Winsock 2 layered service providers. He also lent us his expert debugging skills on our LSP example. Frank Li provided the original RNR example that was the basis for Chapter 10. Frank also developed some of the IP Helper examples included on the CD-ROM. Arvind Murching and Anshul Dhir answered many questions concerning the ATM Winsock provider. Alex Choe (who understands practically every network protocol used today) helped us with the details of the IPX/SPX protocol for our Winsock address family chapter. Ken Evans helped us define and explain many of the IP Helper API data structures defined in Appendix B. Frank Kim and David Mowers helped us understand Windows NT security used in the Windows redirector, mailslots, and named pipe networking technologies. Matt Nibler verified the technical information about the Windows redirector described in Chapter 2. Gary Yukish helped us understand some of the capabilities of RAS. Sachin Kukreja answered many questions about QOS. And finally, Mazahir "Maaaazy" Poonawala provided us with a RAS example for our companion CD-ROM. Thanks also to the Microsoft Platform SDK documentation group for their generous participation and permission.

We also want to thank Rebecca McKay and Donnie Cameron from Microsoft Press for their efforts to ensure that this would be a great book. Rebecca's editing made the book easy to read and understand, while Donnie verified that the technical information was as accurate as possible.

Introduction

Welcome to *Network Programming for Microsoft Windows*! This book covers a wide variety of networking functions available in Windows 95, Windows 98, Windows NT 4, Windows 2000, and Windows CE. The text is designed with the intermediate to advanced programmer in mind, but beginning programmers will find it a useful reference and a comprehensive introduction to the various networking functions.

How to Use This Book

This book is divided into three parts:

- Networking with NetBIOS and the Windows redirector
- Winsock
- Client Remote Access Server (RAS)

We look at NetBIOS in Chapter 1. In our experience in Microsoft Developer Support, we found a number of companies still relying on this technology, and until now there has not been an adequate source of information on how to write NetBIOS applications on Win32. Chapter 1 also offers methods for writing robust and cross-platform applications (especially since many developers use NetBIOS to communicate with legacy systems).

Chapters 2 through 4 cover the Windows redirector, mailslots, and named pipes. As you might know, mailslots and named pipes are based on the redirector. We decided to place the redirector in a separate chapter to give you adequate background information on how these three technologies relate to one another. Mailslots are an unreliable, one-way, message-oriented API that operates regardless of what protocols are available on the system. Named pipes offer more features, such as reliable, two-way, stream- and message-oriented data transmission. Named pipes also offer Windows NT security through the redirector, which no other network API provides.

Part II of this book is dedicated to the Winsock API. Chapter 5 is an introduction to Winsock that covers the most common Winsock protocols that programmers use. All Winsock applications must create a socket in order to perform any network communication. This chapter introduces you to the capabilities of each protocol, while Chapter 6 goes over the specifics of how to create a socket and perform simple name resolution for each protocol type.

Chapter 7 is where the real fun begins. Here we introduce the basic client/server programming model and cover the majority of the Winsock functions that deal with connection establishment and acceptance, data transfers, and more. Chapter 8 follows suit with the different I/O methods offered by Winsock. Since Chapter 7 is meant as a kind of introduction, it discusses only the simplest I/O methods, while Chapter 8 is dedicated to presenting the other I/O methods in detail. If you're new to Winsock, Chapters 5 through 7 should give you a solid background in using the API.

Each remaining chapter in the Winsock section deals with a specific aspect or feature of the API. Socket options and ioctl commands are the subject of Chapter 9. This is where we cover most of the commands that affect the behavior of the socket or the protocol itself. This chapter is useful both for learning purposes and as a reference.

Chapter 10 moves to Winsock 2 registration and name resolution (RNR). This is a protocol-independent method for registering and resolving service names to the underlying protocol's address. The recent release of Windows 2000 and Active Directory makes this a chapter of particular significance.

Chapter 11 covers point-to-multipoint communication, including both IP multicasting and ATM point-to-multipoint. In Chapter 12 we cover Quality of Service (QOS), an exciting new technology that allows for guaranteed network bandwidth allocation to applications. Chapter 13 moves to raw IP sockets. In this chapter, we examine how Winsock applications can use raw IP sockets to use Internet Control Message Protocol (ICMP) and Internet Group Membership Protocol (IGMP) as well as other aspects of raw socket programming.

Chapter 14 covers the Winsock Service Provider Interface (SPI). This interface is a means by which a programmer can install a layer between Winsock and lower-level service providers such as TCP/IP for the purpose of manipulating socket and protocol behavior or name registration and resolution. This is an advanced feature that allows software developers to extend Winsock functionality.

Finally, Chapter 15 discusses the Microsoft Visual Basic Winsock control. We decided to include this chapter after seeing how many developers rely on Visual Basic and this control. The control is limited in its ability to utilize the advanced features of Winsock, but it is fantastic for Visual Basic developers who require simple, easy-to-use network communication.

Part III covers client Remote Access Server (RAS). We decided to include a chapter on RAS because of the popularity of the Internet and dial-up communication. The ability for a programmer to add dial-in capability to a network application is quite useful since it makes the program easier for the user. That is, an end user does not need to know how to establish a dial-up connection to use your network application.

We conclude the book with three appendixes. Appendix A is a NetBIOS command reference that we think you will find invaluable. It straightforwardly lists the required input and output parameters for each command. Appendix B covers new IP Helper functions that provide useful information about the network configuration on the current computer. Appendix C is a Winsock error code reference that describes individual errors in detail and offers possible reasons for their occurrence.

We hope that you will find this book to be a valuable learning tool and reference. We believe it is the most comprehensive book about Windows network programming available.

How to Use the Companion CD-ROM

In each chapter, we present code examples that demonstrate how to use many of the networking APIs we describe. These examples are also available on the accompanying CD-ROM. To install them, place the CD into your drive and Autorun will start the setup program. The setup program can also be accessed manually by running PressCD.exe at the root directory on the disc. The sample code can be installed on your computer, or you can simply access each example from the CD (under Examples\Chapters\Chapter XX).

> NOTE
>
> The CD-ROM requires a computer running a 32-bit Microsoft Windows platform.

In addition to the code examples, the latest version of the Microsoft Platform SDK is included. We included the SDK because many of our examples rely on the latest header files and libraries made available only after Windows 2000 Beta 3.

Support

Every effort has been made to ensure the accuracy of the contents of this book and the companion CD-ROM. Microsoft Press provides corrections for this book at *http://mspress.microsoft.com/support*.

Many of the function definitions and tables in this book were adapted or reprinted here with the generous participation and permission of the Microsoft Platform SDK documentation group. Some material is based on preliminary documentation and is subject to change. For the latest Platform SDK information as well as updates and bug fixes, please visit the MSDN website at *http://msdn.microsoft.com/developer/sdk/platform.asp*.

If you have comments, questions, or ideas regarding this book, please send them to Microsoft Press using postal mail or e-mail:

Microsoft Press
Attn: *Network Programming for Microsoft Windows* editor
One Microsoft Way
Redmond, WA 98052-6399
mspinput@microsoft.com

Part I

Legacy Networking APIs

The first section of this book covers the legacy networking interface NetBIOS, the redirector, and the types of networking communications that operate over the redirector. While the majority of this book is dedicated to Winsock programming, we include Part I because these APIs offer some advantages over Winsock. Chapter 1 covers the NetBIOS interface, which—like Winsock—is a protocol-independent networking API. NetBIOS offers asynchronous calls as well as compatibility with older operating systems such as OS/2, DOS, and so on. Chapter 2 discusses the redirector, which is the basis for the next two topics, mailslots (Chapter 3) and named pipes (Chapter 4). The redirector provides transport-independent file I/O. Mailslots is a simple interface that among other things allows broadcasting and one-way data communication among Windows machines. Finally, named pipes provide a two-way communication channel that also supports Windows security.

Chapter 1

NetBIOS

Network Basic Input/Output System (NetBIOS) is a standard application programming interface (API) developed for IBM in 1983 by Sytek Corporation. NetBIOS defines a programming interface for network communication but doesn't detail how the physical frames are transmitted over a network. In 1985, IBM created the NetBIOS Extended User Interface (NetBEUI), which was integrated with the NetBIOS interface to form an exact protocol. The NetBIOS interface became popular enough that vendors started implementing the NetBIOS programming interface on other protocols such as TCP/IP and IPX/SPX. Platforms and applications throughout the world rely on NetBIOS to this day, including many components of Windows NT, Windows 2000, Windows 95, and Windows 98.

> NOTE
>
> Windows CE does not support the NetBIOS API, even though it supports TCP/IP as a transport protocol and NetBIOS names and name resolution.

The Win32 NetBIOS interface offers backward compatibility with older applications. This chapter discusses the fundamentals of NetBIOS programming. First we will cover the NetBIOS basics, beginning with a discussion of NetBIOS names and LANA numbers. We'll follow this with a discussion of basic services offered by NetBIOS, such as session-oriented and connectionless (datagram) communications. In each section, we will present a simple client and server example. We'll wrap up this chapter with some common pitfalls and bugs that programmers often run into. Appendix A provides a command reference that summarizes each NetBIOS command with its required parameters and a short description of its behavior.

> The OSI Network Model
>
> The Open Systems Interconnect (OSI) model offers a high-level representation of network systems. The OSI model contains seven layers that fully describe fundamental network concepts from the application down to the physical method of data transmissions. Figure 1-1 illustrates the seven layers of the OSI model.

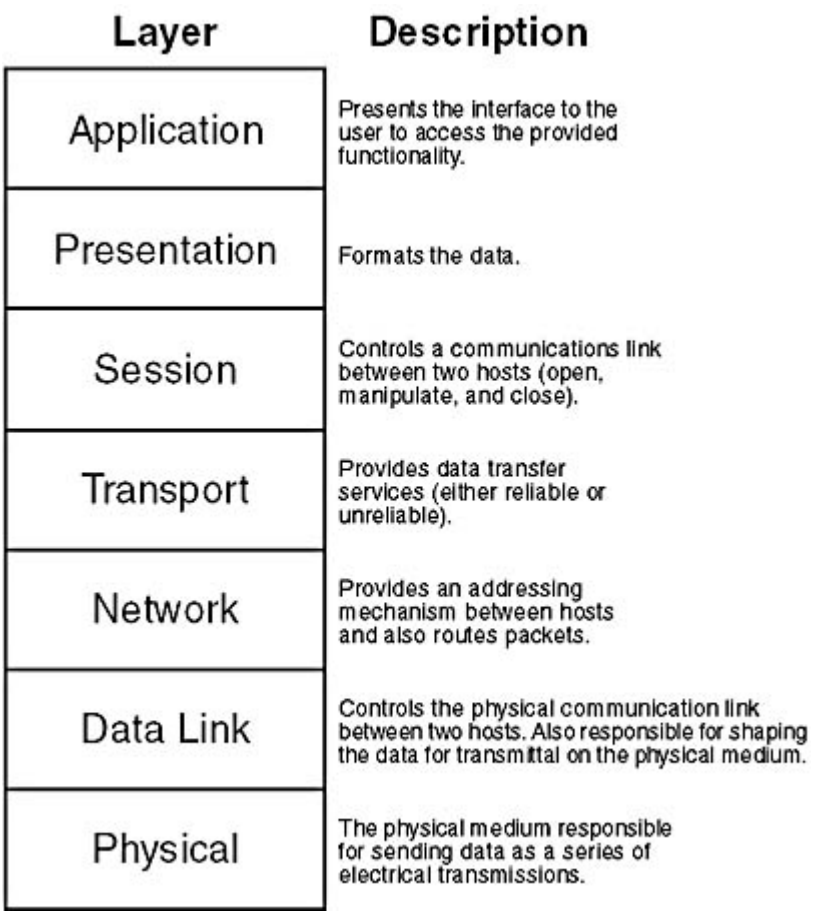

Figure 1-1. *The OSI network model*

Relative to the OSI model, NetBIOS fits primarily into the Session and Transport Layers.

Microsoft NetBIOS

As we mentioned, NetBIOS API implementations exist for numerous network protocols, making the interface protocol-independent. In other words, if you develop your application according to the NetBIOS specification, the application can run over TCP/IP, NetBEUI, or even IPX/SPX. This is a useful feature because it allows a well-written NetBIOS application to run on almost any machine, regardless of the machine's physical network. However, there are a few considerations. For two NetBIOS applications to communicate with each other over the network, they must be running on workstations that have at least one transport protocol in common. For example, if Joe's machine has only TCP/IP installed and Mary's machine has only NetBEUI, NetBIOS applications on Joe's machine won't be able to communicate with applications on Mary's machine.

Additionally, only certain protocols implement a NetBIOS interface. Microsoft TCP/IP and NetBEUI offer a NetBIOS interface by default; however, IPX/SPX does not. Therefore Microsoft provides a version of IPX/SPX that does implement the interface, which is something to keep in mind when designing a network. When installing protocols, the NetBIOS-capable IPX/SPX protocol usually states something to that effect. For example, Windows 2000 offers the protocol NWLink IPX/SPX/NetBIOS Compatible Transport Protocol. In Windows 95 and Windows 98, the IPX/SPX protocol Properties dialog box has a check box that enables NetBIOS over IPX/SPX.

One other important bit of information is that NetBEUI is not a routable protocol. If there is a router between the client machine and the server machine, applications on those machines will not be able to communicate. The router will drop the packets as it receives them. TCP/IP and IPX/SPX are both routable protocols and do not suffer from this limitation. Keep in mind that if you plan to use NetBIOS heavily, you might want to build your networks with at least one of the routable transport protocols. For more information on protocol characteristics and considerations, see Chapter 6.

LANA Numbers

You might be wondering how transport protocols relate to NetBIOS from the programming aspect. The answer is LAN Adapter (LANA) numbers, which are the key to understanding NetBIOS. In the original implementations of NetBIOS, each physical network card was assigned a unique value: a LANA number. Under Win32 this becomes a bit more problematic, as a workstation can have multiple network protocols installed as well as multiple network interface cards.

A LANA number corresponds to the unique pairings of network adapter with transport protocol. For example, if a workstation has two network cards and two NetBIOS-capable transports (such as TCP/IP and NetBEUI), there will be four LANA numbers. The numbers might correspond to the pairings as follows:

0. TCP/IP—Network Card 1
1. NetBEUI—Network Card 1
2. TCP/IP—Network Card 2
3. NetBEUI—Network Card 2

Generally, LANA numbers range from 0 to 9, and the operating system assigns them in no particular order except for LANA 0, which has a special connotation. LANA 0 is the "default" LANA. When NetBIOS first became available, many applications were hardcoded to work only on LANA 0—at that time, most operating systems supported a single LANA number. For the purpose of backward compatibility, you can manually assign LANA 0 to a particular protocol.

In Windows 95 and Windows 98, you can access a network protocol's Properties dialog box through the Network icon in the Control Panel. Select the Configuration tab in the Network dialog box, select a network protocol from the network component list, and click the Properties button. The Advanced tab of the Properties dialog box for each NetBIOS-capable protocol has a Set This Protocol To Be The Default Protocol check box. Selecting the check box rearranges the bindings of the protocols so that the default protocol is assigned LANA 0. Only one protocol can have this check box selected at any given time. Because of the plug-and-play nature of Windows 95 and Windows 98, there is no other way to explicitly set the numbering order for protocols.

Windows NT 4 allows greater flexibility in setting up NetBIOS. The Services tab of the Network dialog box allows you to select NetBIOS Interface from the Network Services list box and click Properties. The NetBIOS Configuration dialog box that appears allows you to specifically assign LANA numbers to each pairing of network

interface with transport protocol. In this dialog box, each network interface is referred to by its driver name; the protocol names are a bit cryptic. Figure 1-2 shows the NetBIOS Configuration dialog box. By clicking the Edit button, you can manually assign the LANA numbers to individual protocols. Windows 2000 also allows you to specifically assign LANA numbers. From the Control Panel, click the Network And Dial-up Connections icon. Then select Advanced Settings from the Advanced menu, and click the LANA numbers tab in the Advanced Settings dialog box.

When developing a robust NetBIOS application, always write code that can handle connections on any LANA number. For example, suppose Mary writes a NetBIOS server application that listens for clients on LANA 2. On Mary's machine, LANA 2 happens to correspond to TCP/IP. Then Joe decides to write a client application to interface with Mary's server, so he decides his application will connect over LANA 2 on his workstation; however, LANA 2 on Joe's machine is NetBEUI. Neither application will ever be able to communicate with the other, even though they both have TCP/IP and NetBEUI installed. To remedy this discrepancy in protocols, Mary's server application should listen for client connections on every available LANA number on Mary's workstation. Likewise, Joe's client application should attempt a connection on each LANA number available on his machine. This way, Mary and Joe can ensure that their applications have the highest degree of success in communication. Of course, writing code that can handle connections on any LANA number does not mean the code will work 100 percent of the time. There remains the case of two machines not having a single common protocol.

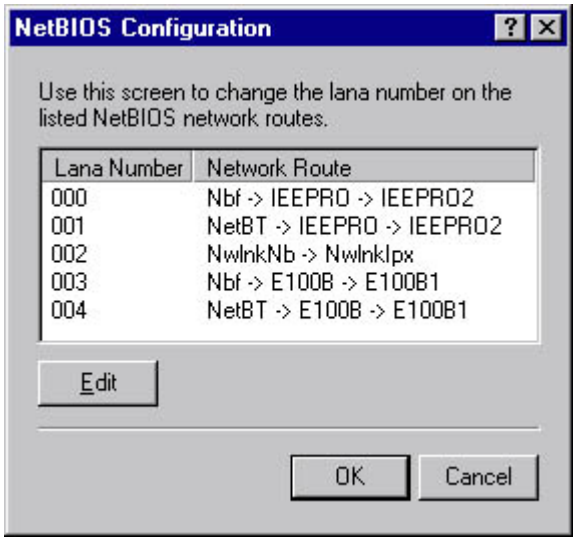

Figure 1-2. *NetBIOS Configuration dialog box. This machine is multihomed with two network cards and three transport protocols: TCP/IP (NetBT), NetBEUI (Nbf), and IPX/SPX (NwlnkNb).*

NetBIOS Names

Now that we know what LANA numbers are, let's move on to NetBIOS names. A process—or application, if you prefer—registers a name on each LANA number that it wants to communicate with. A NetBIOS name is 16 characters long, with the 16th character reserved for special use. When adding a name to the name table, you should initialize the name buffer to spaces. In the Win32 environment, each process has a NetBIOS name table for each available LANA number. Adding a name to LANA 0 means that your application is available to clients connecting on your LANA 0 only. The maximum number of names that can be added to each LANA is 254, with numbering from 1 to 254 (0 and 255 are reserved for the system); however, each operating system sets a default maximum number less than 254 that you can change when resetting each LANA number.

Additionally, there are two types of NetBIOS names: unique and group. A unique name is exactly that: no other process on the network can have that name registered. If another machine does have the name registered, you will receive a duplicate name error. As you might know, machine names in Microsoft networks are NetBIOS names. When a machine boots, it registers its name with the local Windows Internet Naming Server (WINS) server, which reports an error if another machine has that name in use. A WINS server maintains a list of all registered NetBIOS names. Additionally, protocol-specific information can be kept along with the name. For example, on TCP/IP networks, WINS maintains a pairing of NetBIOS names with the IP address that registered the name. If the network is configured without a WINS server, machines perform duplicate name checking by broadcasting a message on the network. If no other machine challenges the message, the network allows the sender to use that name. On the other hand, group names are used to send data to multiple recipients or, conversely, receive data destined for multiple recipients. A group name need not be unique. Group names are used for multicast data transmissions.

The 16th character in NetBIOS names distinguishes most Microsoft networking services. Various networking service and group names are registered with a WINS server by direct name registration from WINS-enabled computers or by broadcast on the local subnet by non-WINS-enabled computers. The Nbtstat command is a utility that you can use to obtain information about NetBIOS names that are registered on the local (or remote) computer. In the example shown in Table 1-1, the Nbtstat -n command produced this list of registered NetBIOS names for user "Davemac" logged on to a computer configured as a primary domain controller and running Windows NT Server with Internet Information Server.

Table 1-1. *NetBIOS name table*

Name	*16th Byte*	*Name Type*	*Service*
DAVEMAC1	<00>	UNIQUE	Workstation service name
DAVEMAC1	<20>	UNIQUE	Server services name
DAVEMACD	<00>	GROUP	Domain name
DAVEMACD	<1C>	GROUP	Domain controller name
DAVEMACD	<1B>	UNIQUE	Master browser name
DAVEMAC1	<03>	UNIQUE	Messenger name
Inet~Services	<1C>	GROUP	Internet Information Server group name
IS~DAVEMAC1	<00>	UNIQUE	Internet Information Server unique name
DAVEMAC1+++++++	<BF>	UNIQUE	Network monitor name

The Nbtstat command is installed only when TCP/IP is an installed protocol. This utility can also query name tables of remote machines by using the -a parameter followed by the remote machine's name, or by using the -A parameter followed by the remote machine's IP address.

Table 1-2 lists the default 16th byte value appended to unique NetBIOS computer names by various Microsoft networking services.

Table 1-2. *Unique name qualifiers*

16th Byte	*Identifies*
<00>	Workstation service name. In general, this is the NetBIOS computer name.
<03>	Messenger service name used when receiving and sending messages. This is the name that is registered with the WINS server as the messenger service on the WINS client and is usually appended to the computer name and to the name of the user currently logged on to the computer.
<1B>	Domain master browser name. This name identifies the primary domain controller and indicates which clients and other browsers to use to contact the domain master browser.
<06>	Remote Access Service (RAS) server service.
<1F>	Network Dynamic Data Exchange (NetDDE) service.
<20>	Server service name used to provide share points for file sharing.
<21>	RAS client.
<BE>	Network Monitor Agent.
<BF>	Network Monitor utility.

Table 1-3 lists the default 16th byte character appended to commonly used NetBIOS group names.

So many qualifiers might seem overwhelming. Think of them as a reference. You probably shouldn't be using them

in your NetBIOS names. To prevent accidental name collisions with your NetBIOS names, you probably want to avoid using the unique name qualifiers. You should be even more careful with group names—no error will be generated if your name collides with an existing group name. When this happens you might start receiving data intended for someone else.

NetBIOS Features

NetBIOS offers both connection-oriented services and connectionless (datagram) services. A connection-oriented service allows two entities to establish a session, or virtual circuit, between them. A session is a two-way communication stream whereby each entity can send the other messages. Session-oriented services provide guaranteed delivery of any data flowing between the two endpoints. In session-oriented services, a server usually registers itself under a certain known name. Clients look for this name in order to communicate with the server. In NetBIOS terms, the server process adds its name to the name table for each LANA it wants to communicate over. Clients on other machines resolve a service name to a machine name and then ask to connect to the server process. As you can see, a few steps are necessary to establish this circuit, and some overhead is involved in first setting up the connection. Session-oriented communication guarantees reliability and packet ordering; however, it is still message-based. That is, if a connected client issues a read command, the server will return only one packet of data on the stream, even if the client supplies a buffer large enough for several packets.

Table 1-3. *Group name qualifiers*

16th Byte	*Identifies*
<1C>	A domain group name that contains a list of the specific addresses of computers that have registered the domain name. The domain controller registers this name. WINS treats this as a domain group: each member of the group must renew its name individually or be released. The domain group is limited to 25 names. When a static 1C name is replicated that clashes with a dynamic 1C name on another WINS server, a union of the members is added, and the record is marked as static. If the record is static, members of the group do not have to renew their IP addresses.
<1D>	The master browser name used by clients to access the master browser. There is one master browser on a subnet. WINS servers return a positive response to domain name registrations but do not store the domain name in their databases. If a computer sends a domain name query to the WINS server, the WINS server returns a negative response. If the computer that sent the domain name query is configured as h-node or m-node, it will then broadcast the name query to resolve the name. The node type refers to how the client attempts to resolve a name. Clients configured for bnode resolution send broadcast packets to advertise and resolve NetBIOS names. The p-node resolution uses point-to-point communication to a WINS server. The m-node resolution is a mix of b-node and p-node in which b-node is used first and then, if necessary, p-node is used. The last resolution method is h-node, or hybrid node. It always attempts to use p-node registration and resolution first, falling back on b-node only upon failure. Windows installations default to h-node.
<1E>	A normal group name. Browsers can broadcast to this name and listen on it to elect a master browser. These broadcasts are for the local subnet and should not cross routers.
<20>	An Internet group name. This type of name is registered with WINS servers to identify groups of computers for administrative purposes. For example, "printersg" could be a registered group name used to identify an administrative group of print servers.
MSBROWSE	Instead of a single appended 16th character, "_MSBROWSE_" is appended to a domain name and broadcast on the local subnet to announce the domain to other master browsers.

On the other hand, there are also connectionless, or datagram, services. In this case a server does register itself under a particular name, but the client simply gathers data and sends it to the network without setting up any connection beforehand. The client addresses the data to the NetBIOS name of the server process. This type of service offers no guarantees, but it offers better performance than connection-oriented services. Furthermore, with datagram services no overhead is involved in setting up a connection. For example, a client might quickly send thousands of bytes of data to a server that crashed two days earlier. The client will never receive any error notifications unless it relies on responses from the server (in which case, it could deduce that something was amiss after not receiving any response for some period of time). Datagram services do not guarantee reliability, nor do they preserve message order.

NetBIOS Programming Basics

Now that we have gone over some of the basic concepts of NetBIOS, we will discuss the NetBIOS API set, which is easy because only one function exists:

```
UCHAR Netbios(PNCB pNCB);
```

All the function declarations, constants, and so on for NetBIOS are defined in the header file Nb30.h. The only library necessary for linking NetBIOS applications is Netapi32.lib. The most important feature of this function is the parameter *pNCB*, which is a pointer to a network control block (NCB). This is a pointer to an *NCB* structure that contains all the information that the required *Netbios* function needs to execute a NetBIOS command. The definition of this structure is as follows:

```
typedef struct _NCB
{
    UCHAR     ncb_command;
    UCHAR     ncb_retcode;
    UCHAR     ncb_lsn;
    UCHAR     ncb_num;
    PUCHAR    ncb_buffer;
    WORD      ncb_length;
    UCHAR     ncb_callname[NCBNAMSZ];
    UCHAR     ncb_name[NCBNAMSZ];
    UCHAR     ncb_rto;
    UCHAR     ncb_sto;
    void      (*ncb_post) (struct _NCB *);
    UCHAR     ncb_lana_num;
    UCHAR     ncb_cmd_cplt;
    UCHAR     ncb_reserve[10];
    HANDLE    ncb_event;
} * PNCB, NCB;
```

Not all members of the structure will be used in every call to NetBIOS; some of the data fields are output parameters (in other words, set on the return from the *Netbios* call). One important tip: it is always a good idea to zero out the *NCB* structure before filling in members prior to a *Netbios* call. Take a look at Table 1-4, which describes the usage of each field. Additionally, the command reference in Appendix A contains a detailed summary of each NetBIOS command and its required (and optional) fields in an *NCB* structure.

Table 1-4. *NCB structure members*

Field	Definition
ncb_command	Specifies the NetBIOS command to execute. Many commands can be executed synchronously or asynchronously by bitwise ORing the *ASYNCH* (0x80) flag and the command.
ncb_retcode	Specifies the return code for the operation. The function sets this value to *NRC_PENDING* while an asynchronous operation is in progress.
ncb_lsn	Identifies the local session number that uniquely identifies a session within the current environment. The function returns a new session number after a successful *NCBCALL* or *NCBLISTEN* command.
ncb_num	Specifies the number of the local network name. A new number is returned for each call with an *NCBADDNAME* or *NCBADDGRNAME* command. You must use a valid number on all datagram commands.
ncb_buffer	Points to the data buffer. For commands that send data, this buffer is the data to send. For commands that receive data, this buffer will hold the data on the return from the *Netbios* function. For other commands, such as *NCBENUM*, the buffer will be the predefined structure *LANA_ENUM*.
ncb_length	Specifies the length of the buffer in bytes. For receive commands, *Netbios* sets this value to the number of bytes received. If the specified buffer is not large enough, *Netbios* returns the error *NRC_BUFLEN*.
ncb_callname	Specifies the name of the remote application.
ncb_name	Specifies the name by which the application is known.
ncb_rto	Specifies the timeout period for receive operations. This value is specified as a multiple of 500-millisecond units. The value 0 implies no timeout. This value is set for *NCBCALL* and *NCBLISTEN* commands that affect subsequent *NCBRECV* commands.
ncb_sto	Specifies the timeout period for send operations. You specify the value in 500-millisecond units. The value 0 implies no timeout. This value is set for *NCBCALL* and *NCBLISTEN* commands that affect subsequent *NCBSEND* and *NCBCHAINSEND* commands.
ncb_post	Specifies the address of the post routine to call upon completion of the asynchronous command. The function is defined as `void CALLBACK PostRoutine(PNCB pncb);` where *pncb* points to the network control block of the completed command.
ncb_lana_num	Specifies the LANA number to execute the command on.
ncb_cmd_cplt	Specifies the return code for the operation. *Netbios* sets this value to *NRC_PENDING* while an asynchronous operation is in progress.
ncb_reserve	Reserved. Must be 0.
ncb_event	Specifies a handle to a Windows event object set to the nonsignaled state. When an asynchronous command is completed, the event is set to its signaled state. Only manual reset events should be used. This field must be 0 if *ncb_command* does not have the *ASYNCH* flag set or if *ncb_post* is nonzero; otherwise, *Netbios* returns the error *NRC_ILLCMD*.

Synchronous vs. Asynchronous

When calling the *Netbios* function, you have the option of making the call synchronous or asynchronous. All NetBIOS commands by themselves are synchronous, which means the call to *Netbios* blocks until the command completes. For an *NCBLISTEN* command, the call to *Netbios* does not return until a client establishes a connection or until an error of some kind occurs. To make a command asynchronous, perform a logical OR of the NetBIOS command with the flag *ASYNCH*. If you specify the *ASYNCH* flag, you must specify either a post routine in the *ncb_post* field or an event handle in the *ncb_event* field. When an asynchronous command is executed, the value

returned from *Netbios* is *NRC_GOODRET* (0x00) but the *ncb_cmd_cplt* field is set to *NRC_PENDING* (0xFF). Additionally, the *Netbios* function sets the *ncb_cmd_cplt* field of the *NCB* structure to *NRC_PENDING* until the command completes. After the command completes, the *ncb_cmd_cplt* field is set to the return value of the command. *Netbios* also sets the *ncb_retcode* field to the return value of the command upon completion.

Common NetBIOS Routines

In this section, we'll examine a basic server NetBIOS application. We will examine the server first because the design of the server dictates how the client should act. Since most servers are designed to handle multiple clients simultaneously, the asynchronous NetBIOS model fits best. We will present server samples using both the asynchronous callback routines and the event model. However, we'll first introduce source code that implements some common functions necessary to most NetBIOS applications. Figure 1-3 is from file Nbcommon.c, which you'll find on the companion disc under *\Examples\Chapter01\Common* . Code examples throughout this book will use functions from this file.

Figure 1-3. *Common NetBIOS routines (Nbcommon.c)*

```
// Nbcommon.c

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#include "nbcommon.h"

//
// Enumerate all LANA numbers
//
int LanaEnum(LANA_ENUM *lenum)
{
    NCB                ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBENUM;
    ncb.ncb_buffer = (PUCHAR)lenum;
    ncb.ncb_length = sizeof(LANA_ENUM);

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBENUM: %d\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Reset each LANA listed in the LANA_ENUM structure. Also, set
// the NetBIOS environment (max sessions, max name table size),
// and use the first NetBIOS name.
//
int ResetAll(LANA_ENUM *lenum, UCHAR ucMaxSession,
             UCHAR ucMaxName, BOOL bFirstName)
{
    NCB                ncb;
    int                i;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBRESET;
```

```
    ncb.ncb_callname[0] = ucMaxSession;
    ncb.ncb_callname[2] = ucMaxName;
    ncb.ncb_callname[3] = (UCHAR)bFirstName;

    for(i = 0; i < lenum->length; i++)
    {
        ncb.ncb_lana_num = lenum->lana[i];
        if (Netbios(&ncb) != NRC_GOODRET)
        {
            printf("ERROR: Netbios: NCBRESET[%d]: %d\n",
                ncb.ncb_lana_num, ncb.ncb_retcode);
            return ncb.ncb_retcode;
        }
    }
    return NRC_GOODRET;
}

//
// Add the given name to the given LANA number. Return the name
// number for the registered name.
//
int AddName(int lana, char *name, int *num)
{
    NCB                ncb;

    ZeroMemory(&ncb, sizeof(NCB));

    ncb.ncb_command = NCBADDNAME;
    ncb.ncb_lana_num = lana;
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBADDNAME[lana=%d;name=%s]: %d\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    *num = ncb.ncb_num;
    return NRC_GOODRET;
}

//
// Add the given NetBIOS group name to the given LANA
// number. Return the name number for the added name.
//
int AddGroupName(int lana, char *name, int *num)
{
    NCB                ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBADDGRNAME;
    ncb.ncb_lana_num = lana;
```

```
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBADDGRNAME[lana=%d;name=%s]: %d\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    *num = ncb.ncb_num;
    return NRC_GOODRET;
}

//
// Delete the given NetBIOS name from the name table associated
// with the LANA number
//
int DelName(int lana, char *name)
{
    NCB                ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBDELNAME;
    ncb.ncb_lana_num = lana;
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBADDNAME[lana=%d;name=%s]: %d\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Send len bytes from the data buffer on the given session (lsn)
// and lana number
//
int Send(int lana, int lsn, char *data, DWORD len)
{
    NCB                ncb;
    int                retcode;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBSEND;
    ncb.ncb_buffer = (PUCHAR)data;
    ncb.ncb_length = len;
    ncb.ncb_lana_num = lana;
    ncb.ncb_lsn = lsn;

    retcode = Netbios(&ncb);
```

```
    return retcode;
}

//
// Receive up to len bytes into the data buffer on the given session
// (lsn) and lana number
//
int Recv(int lana, int lsn, char *buffer, DWORD *len)
{
    NCB                ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBRECV;
    ncb.ncb_buffer = (PUCHAR)buffer;

    ncb.ncb_length = *len;
    ncb.ncb_lana_num = lana;
    ncb.ncb_lsn = lsn;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        *len = -1;
        return ncb.ncb_retcode;
    }
    *len = ncb.ncb_length;

    return NRC_GOODRET;
}
//
// Disconnect the given session on the given lana number
//
int Hangup(int lana, int lsn)
{
    NCB                ncb;
    int                retcode;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBHANGUP;
    ncb.ncb_lsn = lsn;
    ncb.ncb_lana_num = lana;

    retcode = Netbios(&ncb);

    return retcode;
}

//
// Cancel the given asynchronous command denoted in the NCB
// structure parameter
//
int Cancel(PNCB pncb)
{
```

```
    NCB               ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBCANCEL;
    ncb.ncb_buffer = (PUCHAR)pncb;
    ncb.ncb_lana_num = pncb->ncb_lana_num;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: NetBIOS: NCBCANCEL: %d\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Format the given NetBIOS name so that it is printable. Any
// unprintable characters are replaced by a period. The outname
// buffer is the returned string, which is assumed to be at least
// NCBNAMSZ + 1 characters in length.
//
int FormatNetbiosName(char *nbname, char *outname)
{
    int        i;

    strncpy(outname, nbname, NCBNAMSZ);
    outname[NCBNAMSZ - 1] = '\0';
    for(i = 0; i < NCBNAMSZ - 1; i++)
    {
        // If the character isn't printable, replace it with a '.'
        //
        if (!((outname[i] >= 32) && (outname[i] <= 126)))
            outname[i] = '.';
    }
    return NRC_GOODRET;
}
```

The first of the common routines in Nbcommon.c is *LanaEnum* . This is the most basic routine that almost all NetBIOS applications use. This function enumerates the available LANA numbers on a given system. The function initializes an *NCB* structure to 0, sets the *ncb_command* field to *NCBENUM* , assigns a *LANA_ENUM* structure to the *ncb_buffer* field, and sets the *ncb_length* field to the size of the *LANA_ENUM* structure. With the *NCB* structure correctly initialized, the only action that the *LanaEnum* function needs to take to invoke the *NCBENUM* command is to call the *Netbios* function. As you can see, executing a NetBIOS command is fairly easy. For synchronous commands, the return value from *Netbios* will tell you whether the command succeeded. The constant *NRC_GOODRET* always indicates success.

A successful NetBIOS call fills the supplied *LANA_ENUM* structure with the count of LANA numbers on the current machine as well as the actual LANA numbers. The *LANA_ENUM* structure is defined as follows:

```
typedef struct LANA_ENUM
{
    UCHAR   length;
    UCHAR   lana[MAX_LANA + 1];
```

```
} LANA_ENUM, *PLANA_ENUM;
```

The *length* member indicates how many LANA numbers the local machine has. The *lana* field is the array of actual LANA numbers. The value of *length* corresponds to how many elements of the *lana* array will be filled with LANA numbers.

The next function is *ResetAll* . Again, this function will be used in all NetBIOS applications. A well-written NetBIOS program should reset each LANA number that it plans to use. Once you have a *LANA_ENUM* structure with LANA numbers from *LanaEnum* , you can reset them by calling the *NCBRESET* command on each LANA number in the structure. That's exactly what *ResetAll* does; the function's first parameter is a *LANA_ENUM* structure. A reset requires only that the function set *ncb_command* to *NCBRESET* and *ncb_lana_num* to the LANA it needs to reset. Note that while some platforms, such as Windows 95, do not require you to reset each LANA number that you use, it is good practice to do so. Windows NT requires you to reset each LANA number prior to use; otherwise, any other calls to *Netbios* will return error 52 (*NRC_ENVNOTDEF*).

Additionally, when resetting a LANA number you can set certain NetBIOS environment settings via the character fields of *ncb_callname* . *ResetAll* 's other parameters correspond to these environmental settings. The function uses the *ucMaxSession* parameter to set character 0 of *ncb_callname* , which specifies the maximum number of concurrent sessions. Normally, the operating system imposes a default that is less than the maximum. For example, Windows NT 4 defaults to 64 concurrent sessions. *ResetAll* sets character 2 of *ncb_callname* (which specifies the maximum number of NetBIOS names that can be added to each LANA) to the value of the *ucMaxName* parameter. Again, the operating system imposes a default maximum. Finally, *ResetAll* sets character 3, used for NetBIOS clients, to the value of its *bFirstName* parameter. By setting this parameter to *TRUE* , a client uses the machine name as its NetBIOS process name. As a result, a client can connect to a server and send data without allowing any incoming connections. This option is used to save on initialization time because adding a NetBIOS name to the local name table can be costly.

Adding a name to the local name table is another common function. This is what *AddName* does. The parameters are simply the name to add and which LANA number to add it to. Remember that a name table is on a per-LANA basis, and if your application wants to communicate on every available LANA, you need to add the process's name to every LANA. The command for adding a unique name is *NCBADDNAME* . The other required fields are the LANA number to add the name to and the name to add, which must be copied into *ncb_name* . *AddName* initializes the *ncb_name* buffer to spaces first and assumes that the *name* parameter points to a null-terminated string. After adding a name successfully, *Netbios* returns the NetBIOS name number associated with the newly added name in the *ncb_num* field. You use this value with datagrams to identify the originating NetBIOS process. We'll discuss datagrams in greater detail later in this chapter. The most common error encountered when adding a unique name is *NRC_DUPNAME* , which occurs when the name is already in use by another process on the network.

AddGroupName works the same way as *AddName* except that it issues the command *NCBADDGRNAME* and never causes the *NRC_DUPNAME* error.

DelName , another related function, deletes a NetBIOS name from the name table. It requires only the LANA number you want to remove the name from and the name itself.

The next two functions in Figure 1-3, *Send* and *Recv,* are for sending and receiving data in a connected session. These functions are almost identical except for the *ncb_command* field setting. The command field is set to either *NCBSEND* or *NCBRECV* . The LANA number on which to send the data and the session number are both required parameters. A successful *NCBCALL* or *NCBLISTEN* command returns the session number. Clients use the *NCBCALL* command to connect to a known service, while servers use *NCBLISTEN* to "wait" for incoming client connections. When either of these commands succeeds, the NetBIOS interface establishes a session with a unique integer identifier. *Send* and *Recv* also require parameters that map to *ncb_buffer* and *ncb_length* . When sending data, *ncb_buffer* points to the buffer containing the data to send. The length field is the number of characters in the buffer that should be sent. When receiving data, the buffer field points to the block of memory that incoming data is copied to. The length field is the size of the memory chunk. When the *Netbios* function returns, it updates the length field with the number of bytes successfully received. One important aspect of sending data in a session-oriented connection is that a call to the *Send* function will wait until the receiver has posted a *Recv* function. This means that if the sender is pushing a great deal of data and the receiver is not reading it, a lot of resources are being used to buffer the data locally. Therefore, it's a good idea to issue only a few *NCBSEND* or *NCBCHAINSEND* commands simultaneously. To circumvent this problem, use the *Netbios* commands *NCBSENDNA* and *NCBCHAINSENDNA* . With these commands, the sending of the data is performed without waiting for an acknowledgment of receipt from the receiver.

The last two functions near the end of Figure 1-3, *Hangup* and *Cancel* , are for closing established sessions or canceling an outstanding command. You can call the NetBIOS command *NCBHANGUP* to gracefully shut down an established session. When you execute this command, all outstanding receive calls for the given session terminate and return with the session-closed error, *NRC_SCLOSED* (0x0A). If any send commands are outstanding, the

hang-up command blocks until they complete. This delay occurs whether the commands are transferring data or are waiting for the remote side to issue a receive command.

Session Server: Asynchronous Callback Model

Now that we have the basic NetBIOS functions out of the way, we can look at the server that will listen for incoming client connections. Our server will be a simple echo server; it will send back any data that it receives from a connected client. Figure 1-4 contains server code that uses asynchronous callback functions. The code is also available as file Cbnbsvr.c on the companion disc in the /Examples/Chapter01/Server folder. If you look at the function *main* , you will see that first we enumerate the available LANA numbers with *LanaEnum* , and then we reset each LANA with *ResetAll* . Remember that these two steps are generally required of all NetBIOS applications.

Figure 1-4. *Asynchronous callback server (Cbnbsvr.c)*

```
// Cbnbsvr.c

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#include "..\Common\nbcommon.h"

#define MAX_BUFFER      2048
#define SERVER_NAME     "TEST-SERVER-1"

DWORD WINAPI ClientThread(PVOID lpParam);

//
// Function: ListenCallback
//
// Description:
//    This function is called when an asynchronous listen completes.
//    If no error occurred, create a thread to handle the client.
//    Also, post another listen for other client connections.
//
void CALLBACK ListenCallback(PNCB pncb)
{
    HANDLE      hThread;
    DWORD       dwThreadId;

    if (pncb->ncb_retcode != NRC_GOODRET)
    {
        printf("ERROR: ListenCallback: %d\n", pncb->ncb_retcode);
        return;
    }
    Listen(pncb->ncb_lana_num, SERVER_NAME);

    hThread = CreateThread(NULL, 0, ClientThread, (PVOID)pncb, 0,
        &dwThreadId);

    if (hThread == NULL)
    {
        printf("ERROR: CreateThread: %d\n", GetLastError());
        return;
```

```
    }
    CloseHandle(hThread);

    return;
}

//
// Function: ClientThread
//
// Description:
//    The client thread blocks for data sent from clients and
//    simply sends it back to them. This is a continuous loop
//    until the session is closed or an error occurs. If
//    the read or write fails with NRC_SCLOSED, the session
//    has closed gracefully--so exit the loop.
//
DWORD WINAPI ClientThread(PVOID lpParam)
{
    PNCB        pncb = (PNCB)lpParam;
    NCB         ncb;
    char        szRecvBuff[MAX_BUFFER];
    DWORD       dwBufferLen = MAX_BUFFER,
                dwRetVal = NRC_GOODRET;
    char        szClientName[NCBNAMSZ+1];

    FormatNetbiosName(pncb->ncb_callname, szClientName);

    while (1)
    {
        dwBufferLen = MAX_BUFFER;

        dwRetVal = Recv(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, &dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;
        szRecvBuff[dwBufferLen] = 0;
        printf("READ [LANA=%d]: '%s'\n", pncb->ncb_lana_num,
            szRecvBuff);

        dwRetVal = Send(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;
    }

    printf("Client '%s' on LANA %d disconnected\n", szClientName,
        pncb->ncb_lana_num);

    if (dwRetVal != NRC_SCLOSED)
    {
        // Some other error occurred; hang up the connection
        //
        ZeroMemory(&ncb, sizeof(NCB));
```

```
        ncb.ncb_command = NCBHANGUP;
        ncb.ncb_lsn = pncb->ncb_lsn;
        ncb.ncb_lana_num = pncb->ncb_lana_num;

        if (Netbios(&ncb) != NRC_GOODRET)
        {
            printf("ERROR: Netbios: NCBHANGUP: %d\n", ncb.ncb_retcode);
            dwRetVal = ncb.ncb_retcode;
        }
        GlobalFree(pncb);
        return dwRetVal;
    }
    GlobalFree(pncb);
    return NRC_GOODRET;
}

//
// Function: Listen
//
// Description:
//    Post an asynchronous listen with a callback function. Create
//    an NCB structure for use by the callback (since it needs a
//    global scope).
//
int Listen(int lana, char *name)
{
    PNCB        pncb = NULL;

    pncb = (PNCB)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT, sizeof(NCB));
    pncb->ncb_command = NCBLISTEN | ASYNCH;
    pncb->ncb_lana_num = lana;
    pncb->ncb_post = ListenCallback;
    //
    // This is the name clients will connect to
    //
    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_name, name, strlen(name));
    //
    // An '*' means we'll take a client connection from anyone. By
    // specifying an actual name here, we restrict connections to
    // clients with that name only.
    //
    memset(pncb->ncb_callname, ' ', NCBNAMSZ);
    pncb->ncb_callname[0] = '*';

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBLISTEN: %d\n", pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}
```

```
//
// Function: main
//
// Description:
//    Initialize the NetBIOS interface, allocate some resources, add
//    the server name to each LANA, and post an asynch NCBLISTEN on
//    each LANA with the appropriate callback. Then wait for incoming
//    client connections, at which time, spawn a worker thread to
//    handle them. The main thread simply waits while the server
//    threads are handling client requests. You wouldn't do this in a
//    real application, but this sample is for illustrative purposes
//    only.
//
int main(int argc, char **argv)
{
    LANA_ENUM   lenum;
    int         i,
                num;

    // Enumerate all LANAs and reset each one
    //
    if (LanaEnum(&lenum) != NRC_GOODRET)
        return 1;
    if (ResetAll(&lenum, 254, 254, FALSE) != NRC_GOODRET)
        return 1;
    //
    // Add the server name to each LANA, and issue a listen on each
    //
    for(i = 0; i < lenum.length; i++)
    {
        AddName(lenum.lana[i], SERVER_NAME, &num);
        Listen(lenum.lana[i], SERVER_NAME);
    }


    while (1)
    {
        Sleep(5000);
    }
}
```

The next thing that the function *main* does is add your process's name to each LANA number on which you want to accept connections. The server adds its process name, TEST-SERVER-1, to each LANA number in a loop. This is the name the clients will use to connect to our server (padded with spaces, of course). Every character in a NetBIOS name is significant when trying to establish or accept a connection. We can't stress this point enough. Most problems encountered when coding NetBIOS clients and servers involve mismatched names. Be consistent in padding names either with spaces or with some other character. Spaces are the most popular pad character because when they are enumerated and printed out, they are human-readable.

The last and most crucial step for a server is to post a number of *NCBLISTEN* commands. The *Listen* function first allocates an *NCB* structure. When you use asynchronous NetBIOS calls, the *NCB* structure that you submit must persist from the time you issue the call until the call completes. This requires that you either dynamically allocate each *NCB* structure before issuing the command or maintain a global pool of *NCB* structures for use in asynchronous calls. For *NCBLISTEN* , set the LANA number that you want the call to apply to. Note that the code listing in Figure 1-3 logically ORs the *NCBLISTEN* command with the *ASYNCH* command. When specifying the

ASYNCH command, you must make either the *ncb_post* field or the *ncb_event* field nonzero; if you don't, the *Netbios* call will fail with *NRC_ILLCMD* . In Figure 1-4, the *Listen* function sets the *ncb_post* field to our callback function, *ListenCallback* . Next, *Listen* sets the *ncb_name* field to the name of the server process. This is the name that clients will connect to. The function also sets the first character of the *ncb_callname* field to an asterisk (*), signifying that the server will accept a connection from any client. Alternatively, you could place a specific name in the *ncb_callname* field, which would allow only the client who registered that specific name to connect to the server. Finally, *Listen* makes a call to *Netbios* . The call completes immediately, and the *Netbios* function sets the *ncb_cmd_cplt* field of the submitted *NCB* structure to *NRC_PENDING* (0xFF) until the command has completed.

Once *main* resets and posts an *NCBLISTEN* command to each LANA number, the main thread goes into a continuous loop.

> NOTE
>
> Since this server is only a sample, the design is very basic. When writing your own NetBIOS servers, you can do other processing in the main loop or post a synchronous NCBLISTEN in the main loop for one of the LANA numbers.

The callback function executes only when an incoming connection is accepted on a LANA number. When the *NCBLISTEN* command accepts a connection, it calls the function in the *ncb_post* field with the originating *NCB* structure as a parameter. The *ncb_retcode* is set to the return code. Always check this value to see whether the client connection succeeded. A successful connection will result in an *ncb_retcode* of *NRC_GOODRET* (0x00).

If the connection was successful, post another *NCBLISTEN* on the same LANA number. This is necessary because once the original listen succeeds, the server stops listening for client connections on that LANA until another *NCBLISTEN* is submitted. Thus, if your servers require a high availability, you can post multiple *NCBLISTEN* commands on the same LANA so that connections from multiple clients can be accepted simultaneously. Finally, the callback function creates a thread that will service the client. In this example, the thread simply loops and calls a blocking read (*NCBRECV*) followed by a blocking send (*NCBSEND*). The server implements an echo server, which reads messages from connected clients and echoes them back. The client thread loops until the client breaks the connection, at which point the client thread issues an *NCBHANGUP* command to close the connection on its end. From there the client thread frees the *NCB* structure and exits.

For connection-oriented sessions, data is buffered by the underlying protocols, so it is not necessary to always have outstanding receive calls. When a receive command is posted, the *Netbios* function immediately transfers available data to the supplied buffer and the call returns. If no data is available, the receive call blocks until data is present or until the session is disconnected. The same is true for the send command: if the network stack is able either to send data immediately on the wire or to buffer the data in the stack for transmission, the call returns immediately. If the system does not have the buffer space to send the data immediately, the send call blocks until the buffer space becomes available. To circumvent this blocking, you can use the *ASYNCH* command on sends and receives. The buffer supplied to asynchronous sends and receives must have a scope that extends beyond the calling procedure. Another way around blocking sends and receives is to use the *ncb_sto* and *ncb_rto* fields. The *ncb_sto* field is for send timeouts. By specifying a nonzero value, you set an upper limit for how long a send will block before returning. This number is specified in 500-millisecond units. If a command times out, the data is not sent. The same is true of the receive timeout: if no data arrives within the prescribed amount of time, the call returns with no data transferred into the buffers.

Session Server: Asynchronous Event Model

Figure 1-5 illustrates an echo server that is similar to the one in Figure 1-4 but uses Win32 events as the signaling mechanism for completion. The event model is similar to the callback model. The only difference is that with the callback model, the system executes your code when the asynchronous operation completes; while with the event model, your application has to check for the completion of the operation by checking the event status. Because these are standard Win32 events, you can use any of the synchronization routines available, such as *WaitForSingleEvent* and *WaitForMultipleEvents* . The event model is more efficient because it forces the programmer to structure the program to consciously check for completion.

Our event-model server starts out exactly the same as the callback server:

1. Enumerate the LANA numbers.

2. Reset each LANA.

3. Add the server's name to each LANA.

4. Post a listen on each LANA.

The only difference is that you need to keep track of all outstanding listen commands because you must associate event completion with the respective *NCB* blocks that initiate a particular command. The code in Figure 1-5 allocates an array of *NCB* structures equal to the number of LANA numbers (since you want to post one *NCBLISTEN* command on each number). Additionally, the code creates an event for each of the *NCB* structures for signaling the command's completion. The *Listen* function takes one of the *NCB* structures from the array as a parameter.

Figure 1-5. *Asynchronous event server (Evnbsvr.c)*

```
// Evnbsvr.c

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#include "..\Common\nbcommon.h"

#define MAX_SESSIONS    254
#define MAX_NAMES       254

#define MAX_BUFFER      2048
#define SERVER_NAME     "TEST-SERVER-1"

NCB    *g_Clients=NULL;        // Global NCB structure for clients

//
// Function: ClientThread
//
// Description:
//    This thread takes the NCB structure of a connected session
//    and waits for incoming data, which it then sends back to the
//    client until the session is closed
//
DWORD WINAPI ClientThread(PVOID lpParam)
{
    PNCB        pncb = (PNCB)lpParam;
    NCB         ncb;
    char        szRecvBuff[MAX_BUFFER],
                szClientName[NCBNAMSZ + 1];
    DWORD       dwBufferLen = MAX_BUFFER,
                dwRetVal = NRC_GOODRET;

    // Send and receive messages until the session is closed
    //
    FormatNetbiosName(pncb->ncb_callname, szClientName);
    while (1)
    {
        dwBufferLen = MAX_BUFFER;
        dwRetVal = Recv(pncb->ncb_lana_num, pncb->ncb_lsn,
```

```
            szRecvBuff, &dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;

        szRecvBuff[dwBufferLen] = 0;
        printf("READ [LANA=%d]: '%s'\n", pncb->ncb_lana_num,
            szRecvBuff);

        dwRetVal = Send(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;
    }
    printf("Client '%s' on LANA %d disconnected\n", szClientName,
        pncb->ncb_lana_num);
    //
    // If the error returned from a read or a write is NRC_SCLOSED,
    // all is well; otherwise, some other error occurred, so hang up the
    // connection from this side
    //
    if (dwRetVal != NRC_SCLOSED)
    {
        ZeroMemory(&ncb, sizeof(NCB));
        ncb.ncb_command = NCBHANGUP;
        ncb.ncb_lsn = pncb->ncb_lsn;
        ncb.ncb_lana_num = pncb->ncb_lana_num;

        if (Netbios(&ncb) != NRC_GOODRET)
        {

            printf("ERROR: Netbios: NCBHANGUP: %d\n",
                ncb.ncb_retcode);
            GlobalFree(pncb);
            dwRetVal = ncb.ncb_retcode;
        }
    }
    // The NCB structure passed in is dynamically allocated, so
    // delete it before we go
    //
    GlobalFree(pncb);
    return NRC_GOODRET;
}

//
// Function: Listen
//
// Description:
//    Post an asynchronous listen on the given LANA number.
//    The NCB structure passed in already has its ncb_event
//    field set to a valid Windows event handle.
//
int Listen(PNCB pncb, int lana, char *name)
{
```

```
    pncb->ncb_command = NCBLISTEN | ASYNCH;
    pncb->ncb_lana_num = lana;
    //
    // This is the name clients will connect to
    //
    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_name, name, strlen(name));
    //
    // An '*' means we'll accept connections from anyone.
    // We can specify a specific name, which means that only a
    // client with the specified name will be allowed to connect.
    //
    memset(pncb->ncb_callname, ' ', NCBNAMSZ);
    pncb->ncb_callname[0] = '*';

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBLISTEN: %d\n", pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: main
//
// Description:
//    Initialize the NetBIOS interface, allocate some resources, and
//    post asynchronous listens on each LANA using events. Wait for
//    an event to be triggered, and then handle the client
//    connection.
//
int main(int argc, char **argv)
{
    PNCB        pncb=NULL;
    HANDLE      hArray[64],
                hThread;
    DWORD       dwHandleCount=0,
                dwRet,
                dwThreadId;
    int         i,
                num;
    LANA_ENUM   lenum;

    // Enumerate all LANAs and reset each one
    //
    if (LanaEnum(&lenum) != NRC_GOODRET)
        return 1;
    if (ResetAll(&lenum, (UCHAR)MAX_SESSIONS, (UCHAR)MAX_NAMES,
            FALSE) != NRC_GOODRET)
        return 1;
    //
    // Allocate an array of NCB structures (one for each LANA)
```

```
//
g_Clients = (PNCB)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
        sizeof(NCB) * lenum.length);
//
// Create the events, add the server name to each LANA, and issue
// the asynchronous listens on each LANA.
//
for(i = 0; i < lenum.length; i++)
{
    hArray[i] = g_Clients[i].ncb_event = CreateEvent(NULL, TRUE,
        FALSE, NULL);

    AddName(lenum.lana[i], SERVER_NAME, &num);
    Listen(&g_Clients[i], lenum.lana[i], SERVER_NAME);
}
while (1)
{

    // Wait until a client connects
    //
    dwRet = WaitForMultipleObjects(lenum.length, hArray, FALSE,
        INFINITE);
    if (dwRet == WAIT_FAILED)
    {
        printf("ERROR: WaitForMultipleObjects: %d\n",
            GetLastError());
        break;
    }
    // Go through all the NCB structures to see whether more than one
    // succeeded. If ncb_cmd_plt is not NRC_PENDING, there
    // is a client; create a thread, and hand off a new NCB
    // structure to the thread. We need to reuse the original
    // NCB for other client connections.
    //
    for(i = 0; i < lenum.length; i++)
    {
        if (g_Clients[i].ncb_cmd_cplt != NRC_PENDING)
        {
            pncb = (PNCB)GlobalAlloc(GMEM_FIXED, sizeof(NCB));
            memcpy(pncb, &g_Clients[i], sizeof(NCB));
            pncb->ncb_event = 0;

            hThread = CreateThread(NULL, 0, ClientThread,
                (LPVOID)pncb, 0, &dwThreadId);
            CloseHandle(hThread);
            //
            // Reset the handle, and post another listen
            //
            ResetEvent(hArray[i]);
            Listen(&g_Clients[i], lenum.lana[i], SERVER_NAME);
        }
    }
}
```

```
    // Clean up
    //
    for(i = 0; i < lenum.length; i++)
    {
        DelName(lenum.lana[i], SERVER_NAME);
        CloseHandle(hArray[i]);
    }
    GlobalFree(g_Clients);

    return 0;
}
```

The *main* function's first loop cycles through the available LANA numbers, adding the server name and posting the *NCBLISTEN* command to each LANA number, and building an array of event handles. Next call *WaitForMultipleObjects* , which blocks until at least one of the handles becomes signaled. Once one or more of the handles in the even-handle array is in a signaled state, *WaitForMultipleObjects* completes and the code spawns a thread to read incoming messages and send them back to the client. The code creates a copy of the signaled *NCB* structure to pass into the client thread. This is because you want to reuse the original *NCB* to post another *NCBLISTEN* , which you can do by resetting the event and calling *Listen* again on that structure. Note that you don't necessarily have to copy the whole structure. In reality you need only the local session number (*ncb_lsn*) and the LANA number (*ncb_lana_num*). However, the *NCB* structure is a nice container for holding both values to pass into the single parameter of the thread. The client thread used by the event model is the same as the callback model except for the *GlobalFree* statement.

Asynchronous server strategies

Notice that with both servers the possibility exists of a client being denied service. Once the *NCBLISTEN* completes, there is a slight delay until either the callback function is called or the event gets signaled. The servers don't post another *NCBLISTEN* until a few statements later. If the server accepted a client on LANA 2, for example, and then another client attempted a connection before the server issued another *NCBLISTEN* on that LANA, the client would receive the error *NRC_NOCALL* (0x14). This means that the given name had no *NCBLISTEN* posted on it. To avoid this, the server could post multiple *NCBLISTEN* commands on each LANA.

From these two server samples, you can see how easy it is to issue asynchronous commands. The *ASYNCH* flag can be applied to just about any NetBIOS command. Just remember that the *NCB* structure that you pass to *Netbios* must have a global scope.

NetBIOS Session Client

The NetBIOS client is similar in design to the asynchronous event server. Figure 1-6 contains example code for the client. The client performs the familiar routine initialization steps by name. It adds its own name to the name table of each LANA number and then issues an asynchronous connect command. The main loop waits for one of the events to become signaled. At that point, the code cycles through all the *NCB* structures that correspond to the connect commands it issued, one for each LANA. It checks the *ncb_cmd_cplt* status. If it is *NRC_PENDING* , the code cancels the asynchronous command; if the command completed (that is, connected) and the *NCB* doesn't correspond to the *NCB* that was signaled (as specified by the return value from *WaitForMultipleObjects*), the code hangs up the connection. It is possible that if the server is listening on each LANA on its side and the client attempts connections on each of its LANAs, more than one connection can succeed. The code simply closes extra connections with the *NCBHANGUP* command—it needs to communicate over only one channel. By attempting to establish a connection using every LANA on both sides, we allow for the greatest possibility of a successful connection.

Figure 1-6. *Asynchronous event client (Nbclient.c)*

```
// Nbclient.c

#include <windows.h>
#include <stdio.h>
```

```
#include <stdlib.h>

#include "..\Common\nbcommon.h"

#define MAX_SESSIONS     254
#define MAX_NAMES        254

#define MAX_BUFFER       1024

char    szServerName[NCBNAMSZ];

//
// Function: Connect
//
// Description:
//    Post an asynchronous connect on the given LANA number to
//    the server. The NCB structure passed in already has the
//    ncb_event field set to a valid Windows event handle. Just
//    fill in the blanks and make the call.
//
int Connect(PNCB pncb, int lana, char *server, char *client)
{
    pncb->ncb_command = NCBCALL | ASYNCH;
    pncb->ncb_lana_num = lana;

    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_name, client, strlen(client));

    memset(pncb->ncb_callname, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_callname, server, strlen(server));

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBCONNECT: %d\n",
            pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }

    return NRC_GOODRET;
}

//
// Function: main
//
// Description:
//    Initialize the NetBIOS interface, allocate some resources
//    (event handles, a send buffer, and so on), and issue an
//    NCBCALL for each LANA to the given server. Once a connection
//    has been made, cancel or hang up any other outstanding
//    connections. Then send/receive the data. Finally, clean
//    things up.
//
int main(int argc, char **argv)
```

```
{
    HANDLE      *hArray;
    NCB         *pncb;
    char         szSendBuff[MAX_BUFFER];
    DWORD        dwBufferLen,
                 dwRet,
                 dwIndex,
                 dwNum;
    LANA_ENUM    lenum;
    int          i;

    if (argc != 3)
    {
        printf("usage: nbclient CLIENT-NAME SERVER-NAME\n");
        return 1;
    }
    // Enumerate all LANAs and reset each one
    //
    if (LanaEnum(&lenum) != NRC_GOODRET)
        return 1;
    if (ResetAll(&lenum, (UCHAR)MAX_SESSIONS, (UCHAR)MAX_NAMES,
            FALSE) != NRC_GOODRET)
        return 1;
    strcpy(szServerName, argv[2]);
    //
    // Allocate an array of handles to use for asynchronous events.
    // Also allocate an array of NCB structures. We need one handle
    // and one NCB for each LANA number.
    //
    hArray = (HANDLE *)GlobalAlloc(GMEM_FIXED,
        sizeof(HANDLE) * lenum.length);
    pncb   = (NCB *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
        sizeof(NCB) * lenum.length);
    //
    // Create an event, assign it into the corresponding NCB
    // structure, and issue an asynchronous connect (NCBCALL).
    // Additionally, don't forget to add the client's name to each
    // LANA it wants to connect over.
    //
    for(i = 0; i < lenum.length; i++)
    {
        hArray[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
        pncb[i].ncb_event = hArray[i];

        AddName(lenum.lana[i], argv[1], &dwNum);
        Connect(&pncb[i], lenum.lana[i], szServerName, argv[1]);
    }
    // Wait for at least one connection to succeed
    //
    dwIndex = WaitForMultipleObjects(lenum.length, hArray, FALSE,
        INFINITE);
    if (dwIndex == WAIT_FAILED)
    {
```

```
    printf("ERROR: WaitForMultipleObjects: %d\n",
        GetLastError());
}
else
{
    // If more than one connection succeeds, hang up the extra
    // connection. We'll use the connection that was returned
    // by WaitForMultipleObjects. Otherwise, if it's still pending,
    // cancel it.
    //
    for(i = 0; i < lenum.length; i++)
    {
        if (i != dwIndex)
        {
            if (pncb[i].ncb_cmd_cplt == NRC_PENDING)
                Cancel(&pncb[i]);
            else
                Hangup(pncb[i].ncb_lana_num, pncb[i].ncb_lsn);
        }
    }
    printf("Connected on LANA: %d\n", pncb[dwIndex].ncb_lana_num);
    //
    // Send and receive the messages
    //
    for(i = 0; i < 20; i++)
    {
        wsprintf(szSendBuff, "Test message %03d", i);
        dwRet = Send(pncb[dwIndex].ncb_lana_num,
            pncb[dwIndex].ncb_lsn, szSendBuff,
            strlen(szSendBuff));
        if (dwRet != NRC_GOODRET)
            break;
        dwBufferLen = MAX_BUFFER;
        dwRet = Recv(pncb[dwIndex].ncb_lana_num,
            pncb[dwIndex].ncb_lsn, szSendBuff, &dwBufferLen);
        if (dwRet != NRC_GOODRET)
            break;
        szSendBuff[dwBufferLen] = 0;
        printf("Read: '%s'\n", szSendBuff);
    }
    Hangup(pncb[dwIndex].ncb_lana_num, pncb[dwIndex].ncb_lsn);
}
// Clean things up
//
for(i = 0; i < lenum.length; i++)
{
    DelName(lenum.lana[i], argv[1]);
    CloseHandle(hArray[i]);
}
GlobalFree(hArray);
GlobalFree(pncb);

return 0;
```

```
}
```

Datagram Operations

Datagrams are connectionless methods of communication. A sender merely addresses each packet with its destination NetBIOS name and sends it on its way. No checking is performed to ensure data integrity, order of arrival, or reliability.

There are three ways to send a datagram. The first is to direct the datagram at a specific (unique or group) name. This means that only one process can receive that datagram—the process that registered the destination name. The second method is to send a datagram to a group name. Only those processes that registered the given group name will be able to receive the message. Finally, the third way to send a datagram is to broadcast it to the entire network. Any process on any workstation on the local area network can receive the datagram. Sending a datagram to either a unique or a group name uses the *NCBDGSEND* command, while broadcasts use the *NCBDGSENDBC* command.

Using any of the datagram send commands is a simple process. Set the *ncb_num* field to the name number returned from an *NCBADDNAME* command or an *NCBADDGRNAME* command. This is the number that identifies the message sender. Set *ncb_buffer* to the address of the buffer with the data you need to send, and set *ncb_length* to the number of bytes to send. Next, set the *ncb_lana_num* field to the LANA over which you want the datagram to be transmitted. The last step is to set *ncb_callname* to the name of the destination NetBIOS name. This can be a unique name or a group name. If you are sending broadcast datagrams, you perform all of the above steps except for the last one: since all workstations will receive the message, no *ncb_callname* is required.

Of course, in each of the sender scenarios mentioned above, there must be a corresponding datagram receive command to actually receive the data. Datagrams are connectionless; if a datagram reaches a client but the client does not have a receive command already pending, the data is lost and there is no way for the client to recover that data (unless the server resends the data). That is the disadvantage of sending datagrams. However, datagrams are much faster than connection-based methods because they don't require the overhead of error checking, connection setup, and so on.

There are also three methods for receiving datagrams. The first two receive types use the *NCBDGRECV* command. First, you can issue a datagram receive command for messages destined for a specific name—unique or group. Second, you can issue a datagram receive command for any datagram destined for any name in the process's NetBIOS name table. Third, you can issue a receive command for a broadcast datagram with the *NCBDGRECVBC* command.

> NOTE
>
> It is not possible to post a receive command for datagrams destined for a name registered by a different process unless both processes have registered a group name, in which case both processes can receive the same message.

To issue a receive command, set the *ncb_num* field to the name number returned from a successful *NCBADDNAME* or *NCBADDGRNAME* call. This name number specifies which name you are listening on for incoming datagrams. If you set this field to 0xFF, you will receive datagrams destined for any name in this process's NetBIOS name table. Additionally, create a buffer to receive data and set *ncb_buffer* to the buffer's address. Set *ncb_length* to the size of the buffer. Finally, set *ncb_lana_num* to the LANA number on which to wait for datagrams. When a successful call to *Netbios* with the *NCBDGRECV* command (or the *NCBDGRECVBC* command) returns, *ncb_length* will contain the actual number of bytes received and *ncb_callname* will contain the NetBIOS name of the sending process.

The code in Figure 1-7 contains basic datagram functions. All the sends are blocking calls—once the command is issued and the data is on the wire, the function returns, and you shouldn't experience blocking because of data overload. The receive calls are asynchronous events because we do not know what LANA numbers data will come in on. The code is similar to that for the session-oriented server using events. For each LANA, the code posts an asynchronous *NCBDGRECV* (or *NCBDGRECVBC*) and waits until one succeeds, at which point it checks all posted commands, prints the messages for those that succeed, and cancels those commands that are still pending. The example provides functions for both directed and broadcast sends and receives. The program can be compiled into a sample application that can be configured to send or receive datagrams. The program accepts several command line parameters that allow the user to specify the number of datagrams to send or receive, the delay between

sends, the use of broadcasts instead of directed datagrams, the receipt of datagrams for any name, and so on.

Figure 1-7. *NetBIOS datagram sample (Nbdgram.c)*

```
// Nbdgram.c

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#include "..\Common\nbcommon.h"

#define MAX_SESSIONS            254
#define MAX_NAMES               254
#define MAX_DATAGRAM_SIZE       512

BOOL   bSender = FALSE,               // Send or receive datagrams
       bRecvAny = FALSE,              // Receive for any name
       bUniqueName = TRUE,            // Register my name as unique?
       bBroadcast = FALSE,            // Use broadcast datagrams?
       bOneLana = FALSE;              // Use all LANAs or just one?
char   szLocalName[NCBNAMSZ + 1],     // Local NetBIOS name
       szRecipientName[NCBNAMSZ + 1]; // Recipient's NetBIOS name
DWORD  dwNumDatagrams = 25,           // Number of datagrams to send
       dwOneLana,                     // If using one LANA, which one?
       dwDelay = 0;                   // Delay between datagram sends

//
// Function: ValidateArgs
//
// Description:
//    This function parses the command line arguments
//    and sets various global flags indicating the selections
//
void ValidateArgs(int argc, char **argv)
{
    int               i;

    for(i = 1; i < argc; i++)

    {
        if (strlen(argv[i]) < 2)
            continue;
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'n':        // Use a unique name
                    bUniqueName = TRUE;
                    if (strlen(argv[i]) > 2)
                        strcpy(szLocalName, &argv[i][3]);
                    break;
```

```
                case 'g':        // Use a group name
                    bUniqueName = FALSE;
                    if (strlen(argv[i]) > 2)
                        strcpy(szLocalName, &argv[i][3]);
                    break;
                case 's':        // Send datagrams
                    bSender = TRUE;
                    break;
                case 'c':        // # of datagrams to send or receive
                    if (strlen(argv[i]) > 2)
                        dwNumDatagrams = atoi(&argv[i][3]);
                    break;
                case 'r':        // Recipient's name for datagrams
                    if (strlen(argv[i]) > 2)
                        strcpy(szRecipientName, &argv[i][3]);
                    break;
                case 'b':        // Use broadcast datagrams
                    bBroadcast = TRUE;
                    break;
                case 'a':        // Receive datagrams on any name
                    bRecvAny = TRUE;
                    break;
                case 'l':        // Operate on this LANA only
                    bOneLana = TRUE;
                    if (strlen(argv[i]) > 2)
                        dwOneLana = atoi(&argv[i][3]);
                    break;
                case 'd':        // Delay (millisecs) between sends
                    if (strlen(argv[i]) > 2)
                        dwDelay = atoi(&argv[i][3]);
                    break;
                default:
                    printf("usage: nbdgram ?\n");
                    break;
            }
        }
    }
    return;
}

//
// Function: DatagramSend
//
// Description:
//    Send a directed datagram to the specified recipient on the
//    specified LANA number from the given name number to the
//    specified recipient. Also specified is the data buffer and
//    the number of bytes to send.
//
int DatagramSend(int lana, int num, char *recipient,
                 char *buffer, int buflen)
{
    NCB               ncb;
```

```
    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBDGSEND;
    ncb.ncb_lana_num = lana;
    ncb.ncb_num = num;
    ncb.ncb_buffer = (PUCHAR)buffer;
    ncb.ncb_length = buflen;

    memset(ncb.ncb_callname, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_callname, recipient, strlen(recipient));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("Netbios: NCBDGSEND failed: %d\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: DatagramSendBC
//
// Description:
//    Send a broadcast datagram on the specified LANA number from the
//    given name number. Also specified is the data buffer and the number
//    of bytes to send.
//
int DatagramSendBC(int lana, int num, char *buffer, int buflen)
{

    NCB                ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBDGSENDBC;
    ncb.ncb_lana_num = lana;
    ncb.ncb_num = num;
    ncb.ncb_buffer = (PUCHAR)buffer;
    ncb.ncb_length = buflen;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("Netbios: NCBDGSENDBC failed: %d\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: DatagramRecv
//
// Description:
//    Receive a datagram on the given LANA number directed toward the
//    name represented by num. Data is copied into the supplied buffer.
```

```
//    If hEvent is not 0, the receive call is made asynchronously
//    with the supplied event handle. If num is 0xFF, listen for a
//    datagram destined for any NetBIOS name registered by the process.
//
int DatagramRecv(PNCB pncb, int lana, int num, char *buffer,
                 int buflen, HANDLE hEvent)
{
    ZeroMemory(pncb, sizeof(NCB));
    if (hEvent)
    {
        pncb->ncb_command = NCBDGRECV | ASYNCH;
        pncb->ncb_event = hEvent;
    }
    else
        pncb->ncb_command = NCBDGRECV;
    pncb->ncb_lana_num = lana;
    pncb->ncb_num = num;
    pncb->ncb_buffer = (PUCHAR)buffer;
    pncb->ncb_length = buflen;

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("Netbos: NCBDGRECV failed: %d\n", pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: DatagramRecvBC
//
// Description:
//    Receive a broadcast datagram on the given LANA number.
//    Data is copied into the supplied buffer. If hEvent is not 0,
//    the receive call is made asynchronously with the supplied
//    event handle.
//
int DatagramRecvBC(PNCB pncb, int lana, int num, char *buffer,
                   int buflen, HANDLE hEvent)
{
    ZeroMemory(pncb, sizeof(NCB));
    if (hEvent)
    {
        pncb->ncb_command = NCBDGRECVBC | ASYNCH;
        pncb->ncb_event = hEvent;
    }
    else
        pncb->ncb_command = NCBDGRECVBC;
    pncb->ncb_lana_num = lana;
    pncb->ncb_num = num;
    pncb->ncb_buffer = (PUCHAR)buffer;
    pncb->ncb_length = buflen;
```

```
    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("Netbios: NCBDGRECVBC failed: %d\n", pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: main
//
// Description:
//    Initialize the NetBIOS interface, allocate resources, and then
//    send or receive datagrams according to the user's options
//
int main(int argc, char **argv)
{
    LANA_ENUM   lenum;

    int         i, j;
    char        szMessage[MAX_DATAGRAM_SIZE],
                szSender[NCBNAMSZ + 1];
    DWORD      *dwNum = NULL,
                dwBytesRead,
                dwErr;

    ValidateArgs(argc, argv);
    //
    // Enumerate and reset the LANA numbers
    //
    if ((dwErr = LanaEnum(&lenum)) != NRC_GOODRET)
    {
        printf("LanaEnum failed: %d\n", dwErr);
        return 1;
    }
    if ((dwErr = ResetAll(&lenum, (UCHAR)MAX_SESSIONS,
        (UCHAR)MAX_NAMES, FALSE)) != NRC_GOODRET)
    {
        printf("ResetAll failed: %d\n", dwErr);
        return 1;
    }
    //
    // This buffer holds the name number for the NetBIOS name added
    // to each LANA
    //
    dwNum = (DWORD *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
            sizeof(DWORD) * lenum.length);
    if (dwNum == NULL)
    {
        printf("out of memory\n");
        return 1;
    }
    //
```

```
// If we're going to operate on only one LANA, register the name
// on only that specified LANA; otherwise, register it on all
// LANAs
//
if (bOneLana)
{
    if (bUniqueName)
        AddName(dwOneLana, szLocalName, &dwNum[0]);
    else
        AddGroupName(dwOneLana, szLocalName, &dwNum[0]);
}
else
{
    for(i = 0; i < lenum.length; i++)
    {
        if (bUniqueName)
            AddName(lenum.lana[i], szLocalName, &dwNum[i]);
        else
            AddGroupName(lenum.lana[i], szLocalName, &dwNum[i]);
    }
}
// We are sending datagrams
//
if (bSender)
{
    // Broadcast sender
    //
    if (bBroadcast)
    {
        if (bOneLana)
        {
            // Broadcast the message on the one LANA only
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                wsprintf(szMessage,
                    "[%03d] Test broadcast datagram", j);
                if (DatagramSendBC(dwOneLana, dwNum[0],
                    szMessage, strlen(szMessage))
                    != NRC_GOODRET)
                    return 1;
                Sleep(dwDelay);
            }
        }
        else
        {
            // Broadcast the message on every LANA on the local
            // machine
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                for(i = 0; i < lenum.length; i++)
                {
```

```
                    wsprintf(szMessage,
                        "[%03d] Test broadcast datagram", j);
                    if (DatagramSendBC(lenum.lana[i], dwNum[i],
                        szMessage, strlen(szMessage))
                        != NRC_GOODRET)
                        return 1;
                }

                Sleep(dwDelay);
            }
        }
    }
    else
    {
        if (bOneLana)
        {
            // Send a directed message to the one LANA specified
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                wsprintf(szMessage,
                    "[%03d] Test directed datagram", j);
                if (DatagramSend(dwOneLana, dwNum[0],
                    szRecipientName, szMessage,
                    strlen(szMessage)) != NRC_GOODRET)
                    return 1;
                Sleep(dwDelay);
            }
        }
        else
        {
            // Send a directed message to each LANA on the
            // local machine
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                for(i = 0; i < lenum.length; i++)
                {
                    wsprintf(szMessage,
                        "[%03d] Test directed datagram", j);
                    printf("count: %d.%d\n", j,i);
                    if (DatagramSend(lenum.lana[i], dwNum[i],
                        szRecipientName, szMessage,
                        strlen(szMessage)) != NRC_GOODRET)
                        return 1;
                }
                Sleep(dwDelay);
            }
        }
    }
}
else                // We are receiving datagrams
{
```

```
NCB     *ncb=NULL;
char    **szMessageArray = NULL;
HANDLE  *hEvent=NULL;
DWORD   dwRet;

// Allocate an array of NCB structure to submit to each recv
// on each LANA
//
ncb = (NCB *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
            sizeof(NCB) * lenum.length);
//
// Allocate an array of incoming data buffers
//
szMessageArray = (char **)GlobalAlloc(GMEM_FIXED,
        sizeof(char *) * lenum.length);
for(i = 0; i < lenum.length; i++)
    szMessageArray[i] = (char *)GlobalAlloc(GMEM_FIXED,
            MAX_DATAGRAM_SIZE);
//
// Allocate an array of event handles for
// asynchronous receives
//
hEvent = (HANDLE *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
        sizeof(HANDLE) * lenum.length);
for(i = 0; i < lenum.length; i++)
    hEvent[i] = CreateEvent(0, TRUE, FALSE, 0);

if (bBroadcast)
{
    if (bOneLana)
    {
        // Post synchronous broadcast receives on
        // the one LANA specified
        //
        for(j = 0; j < dwNumDatagrams; j++)
        {
            if (DatagramRecvBC(&ncb[0], dwOneLana, dwNum[0],
                szMessageArray[0], MAX_DATAGRAM_SIZE,
                NULL) != NRC_GOODRET)
                return 1;
            FormatNetbiosName(ncb[0].ncb_callname, szSender);
            printf("%03d [LANA %d] Message: '%s' "
                "received from: %s\n", j,
                ncb[0].ncb_lana_num, szMessageArray[0],
                szSender);
        }
    }
    else

    {
        // Post asynchronous broadcast receives on each LANA
        // number available. For each command that succeeded,
        // print the message; otherwise, cancel the command.
```

```
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                for(i = 0; i < lenum.length; i++)
                {
                    dwBytesRead = MAX_DATAGRAM_SIZE;
                    if (DatagramRecvBC(&ncb[i], lenum.lana[i],
                        dwNum[i], szMessageArray[i],
                        MAX_DATAGRAM_SIZE, hEvent[i])
                        != NRC_GOODRET)
                        return 1;
                }
                dwRet = WaitForMultipleObjects(lenum.length,
                    hEvent, FALSE, INFINITE);
                if (dwRet == WAIT_FAILED)
                {
                    printf("WaitForMultipleObjects failed: %d\n",
                        GetLastError());
                    return 1;
                }
                for(i = 0; i < lenum.length; i++)
                {
                    if (ncb[i].ncb_cmd_cplt == NRC_PENDING)
                        Cancel(&ncb[i]);
                    else
                    {
                        ncb[i].ncb_buffer[ncb[i].ncb_length] = 0;
                        FormatNetbiosName(ncb[i].ncb_callname,
                            szSender);
                        printf("%03d [LANA %d] Message: '%s' "
                            "received from: %s\n", j,
                            ncb[i].ncb_lana_num,
                            szMessageArray[i], szSender);
                    }
                    ResetEvent(hEvent[i]);
                }
            }
        }
    }
    else
    {
        if (bOneLana)
        {
            // Make a blocking datagram receive on the specified
            // LANA number
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                if (bRecvAny)
                {
                    // Receive data destined for any NetBIOS name
                    // in this process's name table
                    //
```

```
            if (DatagramRecv(&ncb[0], dwOneLana, 0xFF,
                szMessageArray[0], MAX_DATAGRAM_SIZE,
                NULL) != NRC_GOODRET)
                return 1;
        }
        else
        {
            if (DatagramRecv(&ncb[0], dwOneLana,
                dwNum[0], szMessageArray[0],
                MAX_DATAGRAM_SIZE, NULL)
                != NRC_GOODRET)
                return 1;
        }
        FormatNetbiosName(ncb[0].ncb_callname, szSender);
        printf("%03d [LANA %d] Message: '%s' "
               "received from: %s\n", j,
               ncb[0].ncb_lana_num, szMessageArray[0],
               szSender);
    }
}
else
{
    // Post asynchronous datagram receives on each LANA
    // available. For all those commands that succeeded,
    // print the data; otherwise, cancel the command.
    //
    for(j = 0; j < dwNumDatagrams; j++)
    {
        for(i = 0; i < lenum.length; i++)
        {
            if (bRecvAny)
            {
                // Receive data destined for any NetBIOS
                // name in this process's name table
                //
                if (DatagramRecv(&ncb[i], lenum.lana[i],
                    0xFF, szMessageArray[i],

                    MAX_DATAGRAM_SIZE, hEvent[i])
                    != NRC_GOODRET)
                    return 1;
            }
            else
            {
                if (DatagramRecv(&ncb[i], lenum.lana[i],
                    dwNum[i], szMessageArray[i],
                    MAX_DATAGRAM_SIZE, hEvent[i])
                    != NRC_GOODRET)
                    return 1;
            }
        }
        dwRet = WaitForMultipleObjects(lenum.length,
            hEvent, FALSE, INFINITE);
```

```
                    if (dwRet == WAIT_FAILED)
                    {
                        printf("WaitForMultipleObjects failed: %d\n",
                            GetLastError());
                        return 1;
                    }
                    for(i = 0; i < lenum.length; i++)
                    {
                        if (ncb[i].ncb_cmd_cplt == NRC_PENDING)
                            Cancel(&ncb[i]);
                        else
                        {
                            ncb[i].ncb_buffer[ncb[i].ncb_length] = 0;
                            FormatNetbiosName(ncb[i].ncb_callname,
                                szSender);
                            printf("%03d [LANA %d] Message: '%s' "
                                "from: %s\n", j, ncb[i].ncb_lana_num,
                                szMessageArray[i], szSender);
                        }
                        ResetEvent(hEvent[i]);
                    }
                }
            }
        }
        // Clean up
        //
        for(i = 0; i < lenum.length; i++)
        {
            CloseHandle(hEvent[i]);
            GlobalFree(szMessageArray[i]);
        }
        GlobalFree(hEvent);
        GlobalFree(szMessageArray);
    }
    // Clean things up
    //
    if (bOneLana)
        DelName(dwOneLana, szLocalName);
    else
    {
        for(i = 0; i < lenum.length; i++)
            DelName(lenum.lana[i], szLocalName);
    }
    GlobalFree(dwNum);

    return 0;
}
```

Once you've compiled the example, run the following tests to get an idea of how datagrams work. For learning purposes, you should run two instances of the applications, but on separate machines. If you run them on the same machine, they'll work, but this hides some important concepts. When run on the same machine, the LANA numbers for each side correspond to the same protocol. It's more interesting when they don't. Table 1-5 lists some commands to try. Additionally, Table 1-6 lists all the command line options available for use with the sample

program.

Table 1-5. *Nbdgram.c commands*

Client Command	*Server Command*
Nbdgram /n:CLIENT01	
Nbdgram /s /n:SERVER01 /r:CLIENT01	
Nbdgram /n:CLIENT01 /b	
Nbdgram /s /n:SERVER01 /b	
Nbdgram /g:CLIENTGROUP	
Nbdgram /s /r:CLIENTGROUP	

For the third command, run several clients on various machines. This illustrates one server sending one message to a group, and each member of the group waiting for data will receive the message. Also, try various combinations of the listed commands with the /l:*x* command line option, where *x* is a valid LANA number. This command line option switches the program's mode from performing the commands on all LANAs to performing the commands on the listed LANA only. For example, the command Nbdgram /n:CLIENT01 /l:0 makes the application listen only for incoming datagrams on LANA 0 and ignore any data arriving on any other LANA. Additionally, option /a is meaningful only to the clients. This flag causes the receive command to pick up incoming datagrams destined for any NetBIOS name registered by the process. In our example, this isn't very meaningful because the client registers only one name, but you can at least see how this would be coded. You might want to try modifying the code so that it registers a name for every /n:*name* option in the command line. Start up the server with the recipient flag set to only one of the names that the client registered. The client will receive the data, even though the *NCBDGRECV* command does not specifically refer to a particular name.

Table 1-6. *Command parameters for Nbdgram.c*

Flag	*Meaning*
/n:*my-name*	Register the unique name *my-name* .
/g:*group-name*	Register the group name *group-name* .
/s	Send datagrams (by default, the sample receives datagrams).
/c:*n*	Send or receive *n* number of datagrams.
/r:*receiver*	Specify the NetBIOS name to send the datagrams to.
/b	Use broadcast datagrams.
/a	Post receives for any NetBIOS name (set *ncb_num* to 0xFF).
/l:*n*	Perform all operations on LANA *n* only (by default, all sends and receives are posted on each LANA).
/d:*n*	Wait *n* milliseconds between sends.

Miscellaneous NetBIOS Commands

All of the commands discussed so far deal in some way with setting up a session, sending or receiving data through a session or a datagram, and related subjects. A few commands deal exclusively in getting information. These commands are the adapter status command (*NCBASTAT*) and the find name command (*NCBFINDNAME*), which are discussed in the following sections. The final section deals with matching LANA numbers to their protocols in a programmatic fashion. (This is not actually a NetBIOS function; we'll discuss it because it can gather useful NetBIOS information for you.)

Adapter Status (*NCBASTAT*)

The adapter status command is useful for obtaining information about the local computer and its LANA numbers. Using this command is also the only way to programmatically find the machine's MAC address from Windows 95 and Windows NT 4. With the advent of the IP Helper functions for Windows 2000 and Windows 98 (discussed in Appendix A), there is a more generic interface for finding the MAC address; however, for the other Win32 platforms, using the adapter status command is your only valid option.

The command and its syntax are fairly easy to understand, but two ways of calling the function affect what data is returned. The adapter status command returns an *ADAPTER_STATUS* structure followed by a number of *NAME_BUFFER* structures. The structures are defined as follows:

```
typedef struct _ADAPTER_STATUS {
    UCHAR   adapter_address[6];
    UCHAR   rev_major;
    UCHAR   reserved0;
    UCHAR   adapter_type;
    UCHAR   rev_minor;
    WORD    duration;
    WORD    frmr_recv;
    WORD    frmr_xmit;
    WORD    iframe_recv_err;
    WORD    xmit_aborts;
    DWORD   xmit_success;
    DWORD   recv_success;
    WORD    iframe_xmit_err;
    WORD    recv_buff_unavail;
    WORD    t1_timeouts;
    WORD    ti_timeouts;
    DWORD   reserved1;
    WORD    free_ncbs;
    WORD    max_cfg_ncbs;
    WORD    max_ncbs;
    WORD    xmit_buf_unavail;
    WORD    max_dgram_size;
    WORD    pending_sess;
    WORD    max_cfg_sess;
    WORD    max_sess;
    WORD    max_sess_pkt_size;
    WORD    name_count;
} ADAPTER_STATUS, *PADAPTER_STATUS;
typedef struct _NAME_BUFFER {
    UCHAR   name[NCBNAMSZ];
```

```
    UCHAR   name_num;
    UCHAR   name_flags;
} NAME_BUFFER, *PNAME_BUFFER;
```

The fields of most interest are MAC address (*adapter_address*), maximum datagram size (*max_dgram_size*), and maximum number of sessions (*max_sess*). Also, the *name_count* field tells you how many *NAME_BUFFER* structures were returned. The maximum number of NetBIOS names per LANA is 254, so you have a choice of providing a buffer large enough for all names or calling the adapter status command once with *ncb_length* equal to 0. When the *Netbios* function returns, it provides the necessary buffer size.

The fields required to call *NCBASTAT* are *ncb_command*, *ncb_buffer*, *ncb_length*, *ncb_lana_num*, and *ncb_callname*. If the first character of *ncb_callname* is an asterisk (*), a status command is executed, but only those NetBIOS names added by the calling process are returned. However, if you call *Netbios* with an adapter status command, add a unique name to the current process's name table, and then use that name in the *ncb_callname* field, all NetBIOS names are registered in the local process's name table, as well as any names registered by the system. You can also perform an adapter status query on a machine other than the one the command is executed on. To do this, set the *ncb_callname* field to the machine name of the remote workstation.

> NOTE
>
> Remember that all Microsoft machine names have their 16th byte set to 0 and should be padded with spaces.

The sample program Astat.c is a simple adapter status program that runs the given query on all LANA adapters. Also, by using the /l:LOCALNAME flag, the command is executed on the local machine but provides the entire name table. The /r:REMOTENAME flag executes a remote query for the given machine name.

A few things should be considered when using the adapter status command. First, a multihomed machine will have more than one MAC address. Since NetBIOS provides no way to find which adapters and protocols a LANA is bound to, it is up to you to sort out the values returned. Also, if Remote Access Service (RAS) is installed, the system will allocate LANA numbers for those connections as well. While the RAS connections are unconnected, an adapter status on those LANAs will return all zeros for the MAC address. If a RAS connection is established, the MAC address is the same MAC address RAS assigns to all its virtual network devices. Finally, when you perform a remote adapter status query, you must perform it over a transport protocol that both machines have in common. For example, the system command *Nbtstat* (which is a command line version of *NCBASTAT*) issues its query over the TCP/IP transport only. If the remote machine doesn't have TCP/IP as one of its protocols, the command will fail.

Find Name (*NCBFINDNAME*)

The find name command, available only on Windows NT and Windows 2000, tells you who, if anyone, has a given NetBIOS name registered. In order to perform a successful find name query, the process must add its own unique name to the name table. The required fields for this command are command, LANA number, buffer, and buffer length. The query will return a *FIND_NAME_HEADER* structure and any number of *FIND_NAME_BUFFER* structures, which are defined as follows:

```
typedef struct _FIND_NAME_HEADER {
    WORD    node_count;
    UCHAR   reserved;
    UCHAR   unique_group;
} FIND_NAME_HEADER, *PFIND_NAME_HEADER;

typedef struct _FIND_NAME_BUFFER {
    UCHAR   length;
    UCHAR   access_control;
    UCHAR   frame_control;
    UCHAR   destination_addr[6];
    UCHAR   source_addr[6];
    UCHAR   routing_info[18];
} FIND_NAME_BUFFER, *PFIND_NAME_BUFFER;
```

As with the adapter status command, if the *NCBFINDNAME* command is executed with a buffer length of 0, the *Netbios* function will return the required length with the error *NRC_BUFLEN*.

The *FIND_NAME_HEADER* structure that a successful query returns indicates whether the name is registered as a unique name or a group name. If the field *unique_group* is 0, it is a unique name. The value 1 indicates a group name. The *node_count* field indicates how many *FIND_NAME_BUFFERS* structures were returned. The *FIND_NAME_BUFFER* structure returns quite a bit of information, most of which is useful at the protocol level. However, we're interested in the fields *destination_addr* and *source_addr*. The *source_addr* field contains the MAC address of the network adapter that has registered the name, while the *destination_addr* field contains the MAC address of the adapter that performed the query.

A find name query can be issued on any LANA number on the local machine. The data returned should be identical on all valid LANA numbers for the local network. (For example, you can execute a find name command on a RAS connection to determine whether a name is registered on the remote network.) Under Windows NT 4, you will find the following bug: when a find name query is executed over TCP/IP, *Netbios* returns bogus information. Therefore, if you plan to use this query under Windows NT 4, be sure to pick a LANA corresponding to a transport other than TCP/IP.

Matching Transports to LANA Numbers

This last section discusses matching transport protocols such as TCP/IP and NetBEUI to their LANA numbers. Because there are different potential problems to deal with depending on which transport your application is using, it's nice to be able to find these transports programmatically. This isn't possible with a native NetBIOS call, but it is possible with Winsock 2 under Windows NT 4 and Windows 2000. The Winsock 2 function *WSAEnumProtocols* returns information about available transport protocols. (See Chapters 5 and 6 for more information about *WSAEnumProtocols*.) Although Winsock 2 is available on Windows 95 and by default on Windows 98, the protocol information stored on these platforms does not contain any NetBIOS information, which is what we're looking for.

We won't discuss Winsock 2 in great detail, as this is the subject of Part II of this book. The basic steps involved are loading Winsock 2 through the *WSAStartup* function, calling *WSAEnumProtocols*, and inspecting the *WSAPROTOCOL_INFO* structures returned from the call. The sample Nbproto.c on this book's companion CD contains code for performing this query.

The *WSAEnumProtocols* function takes a buffer to a block of data and a buffer-length parameter. First call the function with a null buffer address and 0 for the length. The call will fail, but the buffer-length parameter will contain the size of the buffer required. Once you have the proper size, call the function again. *WSAEnumProtocols* returns the number of protocols it found. The *WSAPROTOCOL_INFO* structure is large and contains a lot of fields, but the ones we're interested in are *szProtocol*, *iAddressFamily,* and *iProtocol*. If *iAddressFamily* is equal to *AF_NETBIOS*, the absolute value of *iProtocol* is the LANA number for the protocol given in the string *szProtocol*. Additionally, the *ProviderId* GUID can be used to match the returned protocol to certain predefined GUIDs for protocols.

There is only one "gotcha" with this method. Under Windows NT and Windows 2000, the *iProtocol* field for any protocol installed on LANA 0 is the value 0x80000000. This is because protocol 0 is reserved for special use; any protocol assigned LANA 0 will always have the value 0x80000000, so it is a matter of simply checking for this value.

Platform Considerations

Keep these limitations in mind when implementing NetBIOS with the following platforms.

Windows CE

The NetBIOS interface is not available on Windows CE. Although the redirector supports NetBIOS names and name resolution, there is no programming interface support.

Windows 9x

There are several bugs to watch out for in Windows 95 and Windows 98. On any of these two platforms, you must reset all LANA numbers before adding any NetBIOS name to any LANA. This is because resetting one LANA corrupts the name tables of the others; therefore, you want to avoid code similar to the following:

```
LANA_ENUM    lenum;
// Enumerate the LANAs
for(i = 0; i < lenum.length; i++)
{
    Reset(lenum.lana[i]);
    AddName(lenum.lana[i], MY_NETBIOS_NAME);
}
```

Additionally, with Windows 95 do not attempt to perform an asynchronous *NCBRESET* command on the LANA corresponding to the TCP/IP protocol. To begin with, you shouldn't issue this command asynchronously because a reset has to complete before you can do anything with that LANA anyway. If you do decide to execute an *NCBRESET* command asynchronously, your application will cause a fatal error in the NetBIOS TCP/IP virtual device driver (VXD), and you will have to reboot your computer.

General

When performing session-oriented communications, one side can send as much data as it wants; however, the sender really buffers the data it sends until the receiver acknowledges receiving the data by posting a receive command. The NetBIOS commands *NCBSENDNA* and *NCBCHAINSENDNA* are the "no acknowledgment required" versions of the send commands. You can use these commands if you specifically don't want your send commands to wait for acknowledgment from the receiver. Because TCP/IP provides its own acknowledgment scheme in the underlying protocol, these versions of the send commands (versions that don't require acknowledgment from the receiver) behave exactly like the versions that require acknowledgment.

Conclusion

The NetBIOS interface is a powerful but outdated application interface. One of its strengths is its protocol independence—applications can run over TCP/IP, NetBEUI, and SPX/IPX. NetBIOS offers both connection-oriented and connectionless communication. One major advantage the NetBIOS interface has over the Winsock interface is a unified name resolution and registration method. That is, a NetBIOS application only needs a NetBIOS name to operate, whereas a Winsock application that utilizes different protocols needs to be aware of each protocol's addressing scheme (as you'll learn in Part II). Chapter 2 will introduce the redirector. The redirector is an integral part of mailslots and named pipes, which you'll learn about in Chapters 3 and 4.

Chapter 2

The Redirector

Microsoft Windows offers applications the capability to communicate over a network using built-in file system services. This is sometimes referred to as the network operating system (NOS) capability. This chapter explores these networking capabilities using Windows file system components available in Windows 95, Windows 98, Windows NT, Windows 2000, and Windows CE. The purpose of this chapter is to provide an understanding of these capabilities as they relate to the *mailslot* and *named pipe* networking technologies. The mailslot and named pipe networking technologies will be covered in greater detail in Chapters 3 and 4, respectively.

When applications want to access files on a local system, they rely on the operating system to service I/O requests. This is typically referred to as local I/O. For example, when an application opens or closes a file, the operating system determines how to access a device that contains the contents of the specified file. Once the device is found, the I/O request is forwarded to a local device driver. The same operating principle is also available for accessing devices over a network. However, the I/O request must be forwarded over a network to the remote device. This is referred to as I/O redirection. For example, Windows allows you to map or redirect a local disk identifier—such as E:—to a directory share point on a remote computer. When applications reference E:, the operating system redirects the I/O to a device called a *redirector.* The redirector forms a communication channel to a remote computer to access the desired remote directory. This functionality allows applications to use common file system API functions, such as *ReadFile* and *WriteFile*, to access remote files across a network.

This chapter discusses the details of how the redirector is used to redirect I/O requests to remote devices. This is important information—it is the foundation for communication in the mailslot and named pipe technologies. First we will discover how files can be referenced over a network with the Universal Naming Convention (UNC) using the Multiple UNC Provider (MUP) resource locator. This is followed by an explanation of how MUP calls a network provider, which exposes a redirector to form communications among computers using the Server Message Block (SMB) protocol. Finally, we will describe network security considerations when accessing files over a network using basic file I/O operations.

Universal Naming Convention

UNC paths provide a standardized way of accessing files and devices over a network without having to specify or reference a local drive letter that has been mapped to a remote file system. This is important because it allows applications to become drive letter-independent and work seamlessly in a network environment. UNC names are better than names that reference a local drive letter because you don't have to worry about running out of drive letters when forming connections to access server shares. Drive letters also operate on a per-user basis—processes that are not running in your user context cannot access your drive mappings.

UNC names are specified as follows:

```
\\[server]\[share]\[path]
```

The first portion, \\[server], starts with two backslashes followed by a server name. The server name represents a remote server in which an application wants to reference a remote file. The second portion, \[share], represents a share point on the remote server. A share point is simply a directory in a file system that is identified on a network as shared for network user access. The third portion, \[path], represents a directory path to a file in a file system. For example, suppose you have a server named Myserver that contains a directory on a local drive named D:\Myfiles\CoolMusic that is shared out as Myshare. Let's also assume the shared directory contains a file named Sample.mp3. If you would like to reference Sample.mp3 from a remote machine, simply specify the following UNC name:

```
\\Myserver\Myshare\Sample.mp3
```

As you can see, it's much easier to reference a file across a network than it is to map a local drive to the shared directory Myshare.

Referencing files over a network using UNC names hides the details of forming a connection over a network from an application. This is great—a system can easily locate network server directory shares and file paths with UNC names, even over a modem connection. All of the network communication details are handled by a network provider's redirector, which we will discuss later in this chapter. As we will see in Chapters 3 and 4, the mailslot and named pipe technologies depend solely on the use of UNC names for identification.

Figure 2-1 illustrates the common components that form UNC connections on the network operating system in Windows. The figure also shows how the data flows among client and server NOS components. Using the UNC path \\Myserver\Myshare\Sample.mp3 described above, the remainder of this chapter will describe each component and demonstrate what happens when we open this file across a network.

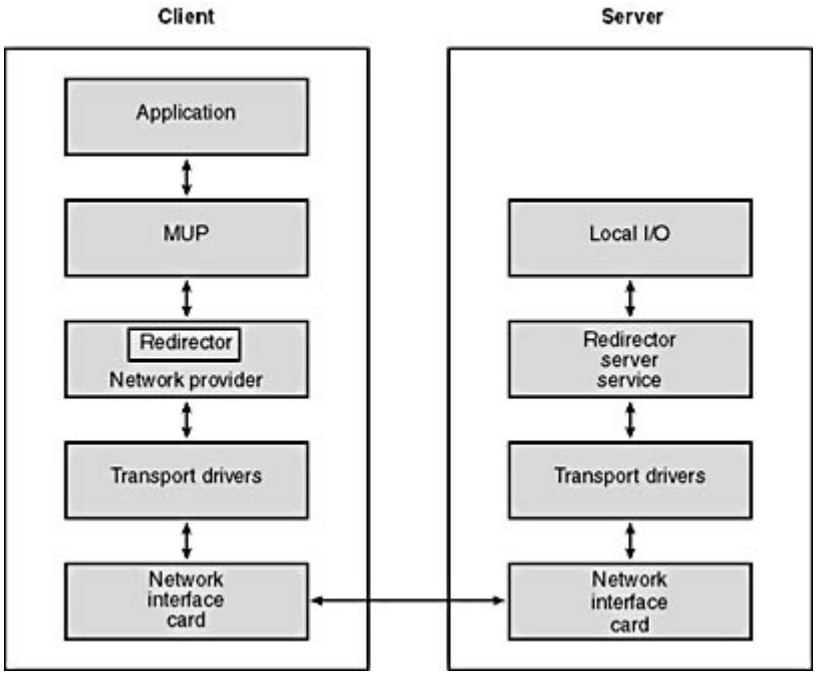

Figure 2-1. *Redirector components*

Multiple UNC Provider

MUP is a resource locator responsible for selecting a network provider to service UNC connections. A network provider is a service that can use network hardware to access resources—such as files and printers—located on a remote computer. MUP uses a network provider to form communications on all UNC name-based I/O requests for files and printers. We will discuss the details of a network provider later in this chapter.

Windows NT, Windows 2000, Windows 95, and Windows 98 are all capable of having multiple network providers installed. For example, Windows platforms provide a network provider named Client for Microsoft Networks. It is also possible to install other third-party network providers, such as Novell's Novell Client v3.01 for Windows 95/98. Thus, more than one network provider might be able to service a single UNC request at a time. On the other hand, Windows CE can have only one network provider: Client for Microsoft Networks.

The primary role of MUP is to decide which network provider should service a UNC request. MUP makes this decision by sending the UNC names in the request to each installed provider (in parallel). If a network provider indicates that it can service a request involving the UNC names, MUP sends the rest of the request to the provider. If more than one provider is capable of servicing a UNC request, MUP chooses the network provider with the most priority. Network provider priority is determined by the order in which providers are installed on your system. In Windows NT, Windows 2000, Windows 95, and Windows 98 the priority can be managed by modifying the registry key ProviderOrder in the following directory in the Windows registry:

```
\HKEY_LOCAL_MACHINE
  \SYSTEM
    \CurrentControlSet
      \Control
        \NetworkProvider
          \Order
```

The installed providers are listed first to last in order of priority. Since Windows CE can have only one provider, it does not use MUP to resolve UNC names. Instead, the UNC requests go directly to its single provider.

Network Providers

As mentioned earlier, a network provider is a service that uses network hardware to access files and printers located on a remote computer. This is considered to be the core function of a network operating system. One of a provider's main capabilities is that of redirecting a local disk identifier—such as E:—to a disk directory located on a remote computer. Providers must also be able to service UNC connection requests. In Windows, network providers do this by exposing a redirector to the operating system.

Windows features a network provider named Client for Microsoft Networks, formally known as the Microsoft Networking Provider (MSNP). The MSNP enables communications among Windows NT 4, Windows 2000, Windows 95, Windows 98, and Windows CE. Windows CE, however, does not have support for multiple network providers and provides only built-in client-side support for the MSNP.

Redirector

A redirector is a component exposed by a network provider to an operating system that accepts and processes remote I/O service requests. It does this by formulating service request messages and sending them to a remote computer's redirector server service. The remote computer's redirector server service receives the request and services it by making local I/O requests. Because a redirector provides I/O services to higher-level services such as MUP, a redirector hides the details of the network layer from applications so that applications don't have to supply protocol-specific parameters to a redirector. Thus, a network provider is protocol-independent: applications can operate in almost any network configuration.

The MSNP provides a redirector that works directly with the networking transport layer and NetBIOS to form communication between a client and a server. The NetBIOS API discussed in Chapter 1 provides a programming interface with these same transports. This redirector provided by MSNP is often referred to as the LAN manager redirector because it is designed around the old Microsoft LAN manager software that provided network operating system capability to MS-DOS applications in the past. (For more detailed information about the NetBIOS programming interface, see Chapter 1.) The NetBIOS interface is capable of communicating over numerous network protocols. This makes the MSNP redirector protocol-independent: your application does not have to concern itself with the specific details of a network protocol. When your application uses the MSNP redirector, it can communicate over TCP/IP, NetBEUI, or even IPX/SPX. This is a rather nice feature because it allows applications to communicate no matter what the physical network comprises. However, one important detail needs to be considered. For two applications to communicate with each other over the network, the two workstations must have at least one transport protocol in common. For example, if workstation A has only TCP/IP installed and workstation B has only IPX installed, the MSNP redirector will not be able to establish communication between the two workstations over a network.

The MSNP redirector communicates with other workstations by sending messages to a remote workstation's redirector server service. These messages are set up in a well-defined structure known as SMB. The actual protocol for how the redirector sends and receives messages to a remote workstation is known as the Server Message Block File Sharing Protocol, or simply the SMB protocol.

Server Message Block

The SMB protocol was originally developed by Microsoft and Intel in the late 1980s to allow remote file systems to be transparently accessed by MS-DOS applications. Today this protocol simply allows a Windows MSNP redirector to communicate with a remote workstation's MSNP server service using an SMB data structure. An SMB data structure contains three basic components: a command code, command-specific parameters, and user data.

The SMB protocol is centered on a simple client-request and server-response messaging model. An MSNP redirector client creates an SMB structure with a specific request indicated in the command code field. If the command requires sending data, such as an SMB Write instruction, data accompanies the request. The SMB structure is then sent over a transport protocol such as TCP/IP to a remote workstation's server service. The remote station's server service processes the client's request and transmits an SMB response data structure back to the client.

Now that we've covered the basics of the components used in forming communication through the MSNP redirector, let's follow how each component communicates when we try to open \\Myserver\Myshare\Sample.mp3 across a network.

1. An application submits a request to the local operating system to open \\Myserver\Myshare\Sample.mp3 using the *CreateFile* API function.
2. The local operating system's file system determines that the I/O request is destined for a remote machine named \\Myserver based on the UNC path description, so it passes the request to MUP.
3. MUP determines that this I/O request is destined for the MSNP provider because the MSNP provider finds \\Myserver on the network using NetBIOS name resolution.
4. The I/O request is passed to the MSNP provider's redirector.
5. The redirector formats the I/O request as an SMB message to open the file Sample.mp3 that is contained in the remote \Myshare directory.
6. The formatted SMB message is transmitted over a network transport protocol.
7. The server named \\Myserver receives the SMB request from the network and passes the request to the server's MSNP redirector server service.
8. The server's redirector server service submits a local I/O request to open the Sample.mp3 file that is located on the \Myshare share point.
9. The server's redirector server service formats an SMB response message regarding the success or failure of the local file open I/O request.
10. The server's SMB response message is sent back to the client over a network transport protocol.
11. The MSNP redirector receives the server's SMB response and passes a return code back to the local operating system.
12. The local operating system returns the return code to the application *CreateFile* API request.

As you can see, the MSNP redirector must go through quite a few steps to grant applications access to remote resources. The MSNP redirector also provides access control to resources on a network. This is a form of network security.

Security

Our discussion of security focuses on accessing resources over a network. However, before we can discuss how security is enforced on resources over a network, we need to discuss security basics on a local machine. Windows NT and Windows 2000 provide the capability to locally and remotely control access to system resources such as files and directories. These resources are considered securable objects. When an application attempts to access a securable object, the operating system checks whether an application has access rights to that object. The three basic access types are read, write, and execute privileges. Windows NT and Windows 2000 accomplish access control through *security descriptors* and *access tokens*.

Security Descriptors

All securable objects contain a security descriptor that defines the access control information associated with an object. A security descriptor consists of a *SECURITY_DESCRIPTOR* structure and its associated security information, which includes the following items:

- Owner Security Identifier (SID) Represents the owner of the object.
- Group SID Represents the primary group owner of the object.
- Discretionary Access Control List (DACL) Specifies who has what type of access to the object. Access types include read, write, and execute privileges.
- System Access Control List (SACL) Specifies the types of access attempts that generate audit records for the object.

Applications cannot directly manipulate the contents of a security descriptor structure. A descriptor can, however, be manipulated indirectly through Win32 security APIs that provide functions for setting and retrieving the security information. We will demonstrate this at the end of this chapter.

Access control lists and access control entities

The DACL and SACL fields of a security descriptor are access control lists (ACLs) that contain zero or more access control entities (ACEs). Each ACE controls or monitors access to an object by a specified user or group. An ACE contains the following type of access control information:

- A SID that identifies the user or the group that the ACE applies to
- A mask that specifies access rights such as read, write, and execute privileges
- A flag that indicates ACE type—allow-access, deny-access, or system-audit

Note that system audit ACE types are used only in SACLs, while allow-access and deny-access ACE types are used in DACLs. Figure 2-2 shows a file object with an associated DACL.

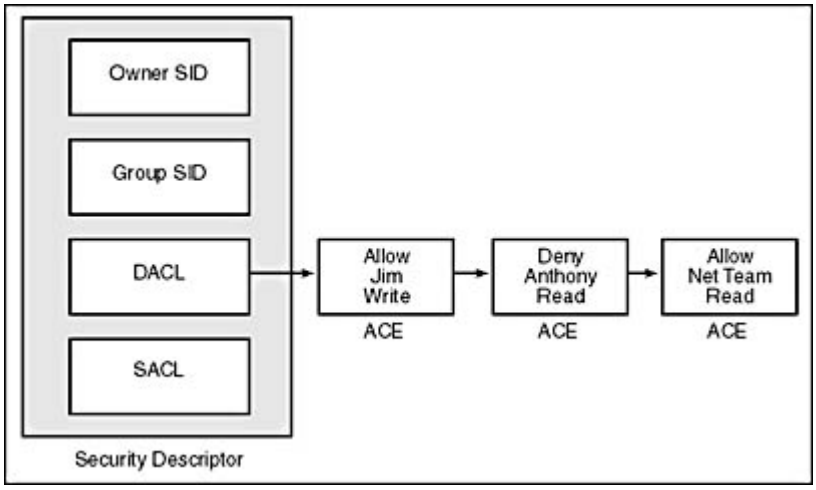

Figure 2-2. *File object with an associated DACL*

If a secured object does not have a DACL (its DACL has been set to a null value using the *SetSecurityDescriptorDacl* API function), the system allows everyone full access. If an object has a DACL, the system allows only the access that is explicitly allowed by the ACEs in the DACL. If there are no ACEs in the DACL, the system does not allow access to anyone. Similarly, if a DACL has some allow-access ACEs, the system implicitly denies access to all users and groups not included in the ACEs.

In most cases, you need to specify only allow-access ACEs, with the following exception: if you include an allow-access ACE for a group, you might have to use deny-access ACEs to exclude members of that particular group. To do this, you must place a user's deny-access ACE ahead of a group's allow-access ACE. Note that the order of the ACL is important because the system reads the ACEs in sequence until access is granted or denied. The user's access-denied ACE must appear first; otherwise, when the system reads the group's access-allowed ACE, it will grant access to the restricted user.

Figure 2-2 shows how to set up a DACL that grants read access to a group named Net Team. Let's assume that the Net Team group consists of Anthony, Jim, and Gary and we want to grant read access to everyone in the group except Anthony. To do this, we must have a deny-access ACE for Anthony set before the allow-access ACE for the Net Team. Figure 2-2 also includes an allow-access ACE to grant Jim write access. Remember that applications do not directly manipulate ACLs; instead, they use security APIs to perform these transactions.

Security identifiers

We have noted that security descriptors and ACEs for securable objects include a SID. A SID is a unique value used to identify a user account, a group account, or a logon session. A security authority, such as a Windows NT server domain, maintains SID information in a security account database. When a user logs on, the system retrieves the user's SID from the database and places it in a user's access token. The system uses the SID in the user's access token to identify the user in all subsequent interactions with Windows NT security.

Access Tokens

When a user logs on to a Windows NT system, the system authenticates the user's account name and password, which are known together as login credentials. If a user logs on successfully, the system creates an access token and assigns it the user's SID. Every process executed on behalf of this user will have a copy of this access token. When a process attempts to access a secured object, the SID in the access token is compared with access rights assigned to SIDs in DACLs.

Network Security

Now that we've briefly explained how security is enforced on a local machine, you are ready to look at security when accessing secured objects over a network. As we saw earlier, the MSNP redirector is responsible for accessing resources among computers. The MSNP redirector is also responsible for establishing a secure link between a client and a server by creating user session credentials.

Session Credentials

There are two types of user credentials: primary login and session credentials. When a user sitting in front of a workstation logs on to the machine, the user name and the password presented by the user become the primary set of credentials and are stored in an access token. Only one set of primary credentials exists at any given time. When a user attempts to establish a connection (either mapping a drive or connecting through UNC names) to a remote resource, the user's primary credentials are used to validate access to the remote resource. Note that with Windows NT and Windows 2000, the user has the option of supplying a different set of credentials to be used in validating with the remote resource. If the user's credentials are valid, the MSNP redirector forms a session between the user's computer and the remote resource. The redirector associates the session with session credentials, which consist of a copy of the credentials the user's computer used to validate the connection with the remote resource. Only one set of session credentials can be established at a time between a user's computer and a remote server. If machine B has two share points, \Hack and \Slash, and if the user of machine A maps \Hack to G and \Slash to H, both sessions share the same session credentials because they both refer to the same remote server.

The MSNP redirector server service handles security access control on a remote server. When the MSNP redirector server attempts to access a secured object, it uses the session credentials to create a remote access token. From there, security is managed as if the access were made locally. Figure 2-3 demonstrates how the MSNP redirector establishes security credentials using Windows NT domain security.

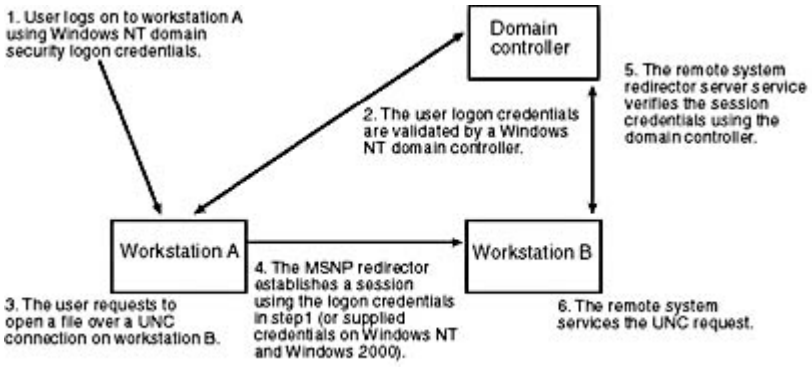

Figure 2-3. *Security credentials demonstration*

A Practical Example

Win32 applications can use the *CreateFile*, *ReadFile*, and *WriteFile* API functions to create, access, and modify files over a network using the MSNP redirector. Windows NT and Windows 2000 are the only platforms that support Win32 security. Figure 2-4 demonstrates how to write a simple application that will create a file over a UNC connection. You will find a file with this code on the companion CD, in the \Examples\Chapter02 directory.

Figure 2-4. *Simple file creation example*

```
#include <windows.h>
#include <stdio.h>

void main(void)
{
    HANDLE FileHandle;
    DWORD BytesWritten;

    // Open a handle to file \\Myserver\Myshare\Sample.txt
    if ((FileHandle = CreateFile("\\\\Myserver\\Myshare\\Sample.txt",
        GENERIC_WRITE | GENERIC_READ,
        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL))
        == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n", GetLastError());
        return;
    }

    // Write 14 bytes to our new file
    if (WriteFile(FileHandle, "This is a test", 14,
        &BytesWritten, NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastError());
        return;
    }

    if (CloseHandle(FileHandle) == 0)
    {
        printf("CloseHandle failed with error %d\n", GetLastError());
        return;
    }
}
```

Conclusion

This chapter introduced you to the Windows redirector, which enables an application to access Windows file system resources over a network. We explained the fundamental way in which the redirector communicates over a network, followed by a discussion of security features offered by Windows NT and Windows 2000 when applications use the redirector. The next two chapters cover the mailslot and named pipe technologies, which depend solely on the redirector for all network communications.

Chapter 3

Mailslots

Microsoft Windows NT, Windows 2000, Windows 95, and Windows 98 (but not Windows CE) include a simple one-way interprocess communication (IPC) mechanism known as *mailslots*. In simplest terms, mailslots allow a client process to transmit or broadcast messages to one or more server processes. Mailslots can assist transmission of messages among processes on the same computer or among processes on different computers across a network. Developing applications using mailslots is simple, requiring no formal knowledge of underlying network transport protocols such as TCP/IP or IPX. Because mailslots are designed around a broadcast architecture, you can't expect reliable data transmissions via mailslots. They can be useful, nevertheless, in certain types of network programming situations in which delivery of data isn't mission-critical.

One possible scenario for using mailslots is developing a messaging system that includes everyone in your office. Imagine that your office environment has a large number of workstations. Your office is suffering from a soda shortage, and every workstation user in your office is interested in knowing every few minutes how many Cokes are available in the soda machine. Mailslots lend themselves well to this type of situation. You can easily implement a mailslot client application that monitors the soda count and broadcasts to every interested workstation user the total number of available Cokes at five-minute intervals. Because mailslots don't guarantee delivery of a broadcast message, some workstation users might not receive all updates. A few failures of transmission won't be a problem in this case because messages sent at five-minute intervals with occasional misses are still frequent enough to keep the workstation users well informed.

The major limitation of mailslots is that they permit only unreliable one-way data communication from a client to a server. The biggest advantage of mailslots is that they allow a client application to easily send broadcast messages to one or more server applications.

This chapter explains how to develop a mailslot client/server application. We'll describe mailslot naming conventions before we address the message sizing considerations that control the overall behavior of mailslots. Next we'll show you the details of developing a basic client/server application. At the end of this chapter, we'll tell you about known problems and limitations of mailslots and offer workaround solutions.

Mailslot Implementation Details

Mailslots are designed around the Windows file system interface. Client and server applications use standard Win32 file system I/O functions, such as *ReadFile* and *WriteFile*, to send and receive data on a mailslot and take advantage of Win32 file system naming conventions. Mailslots rely on the Windows redirector to create and identify mailslots using a file system named the Mailslot File System (MSFS). Chapter 2 described the Windows redirector in greater detail.

Mailslot Names

Mailslots use the following naming convention for identification:

```
\\server\Mailslot\[path]name
```

The string above is divided into three portions: \\server, \Mailslot, and \[path]name. The first string portion, \\server, represents the name of the server on which a mailslot is created and on which a server application is running. The second portion, \Mailslot, is a hardcoded mandatory string for notifying the system that this filename belongs to MSFS. The third portion, \[path]name, allows applications to uniquely define and identify a mailslot name; the path portion might specify multiple levels of directories. For example, the following types of names are legal for identifying a mailslot:

```
\\Oreo\Mailslot\Mymailslot
\\Testserver\Mailslot\Cooldirectory\Funtest\Anothermailslot
\\.\Mailslot\Easymailslot
\\*\Mailslot\Myslot
```

The server string portion can be represented as a dot (.), an asterisk (*), a domain name, or a server name. A domain is simply a group of workstations and servers that share a common group name. We'll examine mailslot names in greater detail later in this chapter, when we cover implementation details of a simple client.

Because mailslots rely on the Windows file system services for creation and transferring data over a network, the interface protocol is independent. When creating your application, you don't have to worry about the details of underlying network transport protocols to form communications among processes across a network. When mailslots communicate remotely to computers across a network, the Windows file system services rely on the Windows redirector to send data from a client to a server using the Server Message Block (SMB) protocol. Messages are typically sent via connectionless transfers, but you can force the Windows redirector to use connection-oriented transfers on Windows NT and Windows 2000, depending on the size of your message.

Message Sizing

Mailslots normally use *datagrams* to transmit messages over a network. Datagrams are small packets of data that are transmitted over a network in a connectionless manner. Connectionless transmission means that each data packet is sent to a recipient without packet acknowledgment. This is unreliable data transmission, which is bad in that you cannot guarantee message delivery. However, connectionless transmission does give you the capability to broadcast a message from one client to many servers. The exception to this occurs on Windows NT and Windows 2000 when messages exceed 424 bytes.

On Windows NT and Windows 2000, messages larger than 426 bytes are transferred using a connection-oriented protocol over an SMB session instead of using datagrams. This allows large messages to be transferred reliably and efficiently. However, you lose the ability to broadcast a message from a client to many servers. Connection-oriented transfers are limited to one-to-one communication: one client to one server. Connection-oriented transfers normally provide reliable guaranteed delivery of data between processes, but the mailslot interface on

Windows NT and Windows 2000 does not guarantee that a message will actually be written to a mailslot. For example, if you send a large message from a client to a server that does not exist on a network, the mailslot interface does not tell your client application that it failed to submit data to the server. Since Windows NT and Windows 2000 change their transmission method based on message size, an interoperability problem occurs when you send large messages between a machine running Windows NT or Windows 2000 and a machine running Windows 95 or Windows 98.

Windows 95 and Windows 98 deliver messages via datagrams only, regardless of message size. If a Windows 95 or Windows 98 client attempts to send a message larger than 424 bytes to a Windows NT or Windows 2000 server, Windows NT and Windows 2000 accept the first 424 bytes and truncate the remaining data. Windows NT and Windows 2000 expect larger messages to be sent over a connection-oriented SMB session. A similar problem exists in transferring messages from a Windows NT or Windows 2000 client to a Windows 95 or Windows 98 server. Remember that Windows 95 and Windows 98 receive data via datagrams only. Because Windows NT and Windows 2000 transfer data via datagrams for messages 426 bytes or smaller, Windows 95 and Windows 98 cannot receive messages larger than 426 bytes from such clients. Table 3-1 outlines these message size limitations in detail.

> NOTE
>
> Windows CE was intentionally left out of Table 3-1 because the mailslot-programming interface is not available. Also note that messages sized 425 to 426 bytes are not listed in this table due to a Windows NT and Windows 2000 redirector limitation.

Table 3-1. *Mailslot message size limitations*

Transfer Direction	Connectionless Transfer via Datagrams	Connection-Oriented Transfer
Windows 95 or Windows 98 -> Windows 95 or Windows 98	Message size up to 64 KB.	Not supported
Windows NT or Windows 2000 -> Windows NT or Windows 2000	Messages must be 424 bytes or less.	Messages must be greater than 426 bytes
Windows NT or Windows 2000 -> Windows 95 or Windows 98	Messages must be 424 bytes or less.	Not supported
Windows 95 or Windows 98 -> Windows NT or Windows 2000	Messages must be 424 bytes or less; otherwise, the message is truncated.	Not supported

Another limitation of Windows NT and Windows 2000 is worth discussion because it affects datagram data transmissions. The Windows NT and Windows 2000 redirector cannot send or receive a complete datagram message sized 425 or 426 bytes. For example, if you send out a message from a Windows NT or Windows 2000 client to a Windows 95, Windows 98, Windows NT, or Windows 2000 server, the Windows NT redirector truncates the message to 424 bytes before the sending it to the destination server.

To accomplish total interoperability among all Windows platforms, we strongly recommend limiting message sizes to 424 bytes or less. If you are looking for connection-oriented transfers, consider using named pipes instead of mailslots. Named pipes are covered in Chapter 4.

Compiling Applications

When you build a mailslot client or server application using Microsoft Visual C++, your application must include the Winbase.h include file in your program files. If you include Windows.h (as most applications do) you can omit Winbase.h. Your application is also responsible for linking with Kernel32.lib, which is typically configured with the Visual C++ linker flags.

Error Codes

All Win32 API functions that are used in developing mailslot client and server applications (except for *CreateFile* and *CreateMailslot*) return the value 0 when they fail. The *CreateFile* and *CreateMailslot* API functions return *INVALID_HANDLE_VALUE*. When these API functions fail, applications should call the *GetLastError* function to retrieve specific information about the failure. For a complete list of error codes, see the standard Windows error codes in Appendix C or consult the header file Winerror.h.

Basic Client/Server

As we mentioned earlier, mailslots feature a simple client/server design architecture in which data can flow only from a client to a server. The data communication model is one-way, or unidirectional. The server is responsible for creating a mailslot and is the only process that can read data from it. Mailslot clients are processes that open instances of mailslots and are the only processes that can write data to them.

Mailslot Server Details

Implementing a mailslot requires developing a server application to create a mailslot. The following steps describe how to write a basic server application:

1. Create a mailslot handle using the *CreateMailslot* API function.
2. Receive data from any client by calling the *ReadFile* API function using the mailslot handle.
3. Close the mailslot handle using the *CloseHandle* API function.

As you can see, very few API calls are needed to develop a mailslot server application.

Server processes create mailslots using the *CreateMailslot* API call, which is defined as

```
HANDLE CreateMailslot(
    LPCTSTR lpName,
    DWORD nMaxMessageSize,
    DWORD lReadTimeout,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

The first parameter, *lpName*, specifies the name of the mailslot. The name must have the following form:

```
\\.\Mailslot\[path]name
```

Notice that the server name is represented as a dot, which represents the local machine. This is required because you cannot create a mailslot on a remote computer. In the *lpName* parameter, name must represent a unique name. This might simply be a name, or a full directory path might precede it.

The *nMaxMessageSize* parameter defines the maximum size—in bytes—of a message that can be written to a mailslot. If a client writes more than *nMaxMessageSize* bytes, the server doesn't see the message. Specifying the value 0 allows the server to accept a message of any size.

Read operations can operate in blocking or nonblocking mode on a mailslot, depending on the *lReadTimeout* parameter, which determines the amount of time in milliseconds that read operations wait for incoming messages. Specifying the value *MAILSLOT_WAIT_FOREVER* allows read operations to block and wait indefinitely until incoming data is available to be read. If you specify 0, read operations return immediately. We will discuss details of reading later in this chapter. The *lpSecurityAttributes* parameter determines access control rights to a mailslot. On Windows 95 and 98, this parameter must be *NULL* because you cannot apply security to objects. On Windows NT and Windows 2000, this parameter is only partially implemented, so you should also specify a *NULL* parameter. The only security that you can enforce on a mailslot is for local I/O, in which a client attempts to open a mailslot with a dot (.) for the server name. A client can get around this security by specifying the server's actual name instead of a dot (.), as when making a remote I/O call. The *lpSecurityAttributes* parameter is not implemented for remote I/O on Windows NT and Windows 2000 because of the extreme inefficiency of forming an authenticated

session between the client and the server every time a message is sent. Mailslots, therefore, only partially follow the Windows NT and Windows 2000 security model found in the standard file systems. As a consequence, any mailslot client on your network can send data to your server.

After a mailslot is created with a valid handle, you can begin reading data. The server is the only process that can read data from a mailslot. The server should use the Win32 *ReadFile* function to accomplish this. *ReadFile* is defined as

```
BOOL ReadFile(
    HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead,
    LPDWORD lpNumberOfBytesRead,
    LPOVERLAPPED lpOverlapped
);
```

CreateMailslot returns the handle *hFile*. The *lpBuffer* and *nNumberOfBytesToRead* parameters determine how much data can be read off a mailslot. It is important to make the size of this buffer greater than the *nMaxMessageSize* parameter from the *CreateMailslot* API call. Additionally, the buffer must be larger than incoming messages on the mailslot; if it is not larger, *ReadFile* will fail with the error *ERROR_INSUFFICIENT_BUFFER*. The *lpNumberOfBytesRead* parameter reports the actual number of bytes read when the *ReadFile* operation completes.

The *lpOverlapped* parameter provides a way to read data asynchronously off a mailslot. This parameter uses the Win32 overlapped I/O mechanism, which we describe in greater detail in Chapter 4. By default, the *ReadFile* operation blocks (waits) on I/O until data is available for reading. Overlapped I/O can be accomplished only on Windows NT and Windows 2000; you should specify *NULL* for this parameter when using Windows 95 or Windows 98. Figure 3-1 further demonstrates how to write a simple mailslot server application.

Figure 3-1. *Server mailslot example*

```
// Server1.cpp

#include <windows.h>
#include <stdio.h>

void main(void)
{
    HANDLE Mailslot;
    char buffer[256];
    DWORD NumberOfBytesRead;

    // Create the mailslot
    if ((Mailslot = CreateMailslot("\\\\.\\Mailslot\\Myslot", 0,
        MAILSLOT_WAIT_FOREVER, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("Failed to create a mailslot %d\n", GetLastError());
        return;
    }

    // Read data from the mailslot forever!
    while(ReadFile(Mailslot, buffer, 256, &NumberOfBytesRead,
        NULL) != 0)
    {
```

```
        printf("%.*s\n", NumberOfBytesRead, buffer);
    }
}
```

Mailslot Client Details

Implementing a client requires developing an application to reference and write to an existing mailslot. The following steps describe how to write a basic client application:

1. Open a reference handle to the mailslot we want to send data to using the *CreateFile* API.
2. Write data to the mailslot by calling the *WriteFile* API.
3. Once you are finished writing data, close the mailslot handle using the *CloseHandle* API.

As we described earlier, mailslot clients communicate to mailslot servers in a connectionless manner. When a client opens a reference handle to a mailslot, the client does not form a connection to the mailslot server. Mailslots are referenced by using the *CreateFile* API call, which is defined as

```
HANDLE CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
```

The *lpFileName* parameter describes one or more mailslots that can be written to by using the mailslot name format described earlier in this chapter. Table 3-2 describes mailslot naming conventions in greater detail. The *dwDesiredAccess* parameter must be set to *GENERIC_WRITE* because a client can only write data to the server. The *dwShareMode* parameter must be set to *FILE_SHARE_READ*, allowing the server to open and perform read operations on the mailslot. The *lpSecurityAttributes* parameter has no effect on mailslots and should be set to *NULL*. The *dwCreationDisposition* flag should be set to *OPEN_EXISTING*. This setting is useful when a client and a server are operating on the same machine: If the server has not created the mailslot, the *CreateFile* API function fails. The *dwCreationDisposition* parameter has no effect if the server is operating remotely. The *dwFlagsAndAttributes* parameter should be defined as *FILE_ATTRIBUTE_NORMAL*. The *hTemplateFile* parameter should be set to *NULL*.

Table 3-2. *Mailslot name types*

Name Format	*Description*
\\.\mailslot\name	Identifies a local mailslot on the same machine
\\servername\mailslot\name	Identifies a remote mailslot server named *servername*
\\domainname\mailslot\name	Identifies all mailslots of a particular *name* in the specified *domain*
*\mailslot\name	Identifies all mailslots of a particular *name* in the system's primary domain

After a handle has been successfully created, you can begin writing data to a mailslot. Remember, a client can only write data to the mailslot. This can be accomplished using the Win32 *WriteFile* function, defined as

```
BOOL WriteFile(
    HANDLE hFile,
    LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPDWORD lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped
);
```

The *hFile* parameter is the reference handle that *CreateFile* returns. The *lpBuffer* and *nNumberOfBytesToWrite* parameters determine how many bytes will be sent from the client to the server. The maximum size of a message is 64 KB. If the mailslot handle was created using a domain or asterisk format, the message size is limited to 424 bytes on Windows NT and Windows 2000 and 64 KB on Windows 95 and Windows 98. If a client attempts to send a message that exceeds those limits, the *WriteFile* function fails and the *GetLastError* function returns *ERROR_BAD_NETPATH*. This happens because the message is sent as a broadcast datagram to all servers on the network. The *lpNumberOfBytesWritten* parameter returns the number of bytes sent to a server when the *WriteFile* operation completes.

The *lpOverlapped* parameter provides a way to write data asynchronously to a mailslot. Since mailslots feature connectionless data transfer, the *WriteFile* function is not subject to blocking on I/O calls. This parameter should be set to *NULL* on the client. Figure 3-2 further demonstrates how to write a simple mailslot client application.

Figure 3-2. *Client mailslot example*

```
// Client.cpp

#include <windows.h>
#include <stdio.h>

void main(int argc, char *argv[])
{
    HANDLE Mailslot;
    DWORD BytesWritten;
    CHAR ServerName[256];

    // Accept a command line argument for the server to send
    // a message to
    if (argc < 2)
    {
        printf("Usage: client <server name>\n");
        return;
    }

    sprintf(ServerName, "\\\\%s\\Mailslot\\Myslot", argv[1]);

    if ((Mailslot = CreateFile(ServerName, GENERIC_WRITE,
        FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
        NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n", GetLastError());
        return;
    }
```

```
    if (WriteFile(Mailslot, "This is a test", 14, &BytesWritten,
        NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastError());
        return;
    }

    printf("Wrote %d bytes\n", BytesWritten);

    CloseHandle(Mailslot);
}
```

Additional Mailslot APIs

A mailslot server application can use two additional API functions to interact with a mailslot: *GetMailslotInfo* and *SetMailslotInfo*. The *GetMailslotInfo* function retrieves message sizing information when messages become available on a mailslot. Applications can use this to dynamically adjust their buffers for incoming messages of varying length. *GetMailslotInfo* can also be used to poll for incoming data. *GetMailslotInfo* is defined as

```
BOOL GetMailslotInfo(
    HANDLE hMailslot,
    LPDWORD lpMaxMessageSize,
    LPDWORD lpNextSize,
    LPDWORD lpMessageCount,
    LPDWORD lpReadTimeout
);
```

The *hMailslot* parameter identifies a mailslot returned from the *CreateMailslot* API call. The *lpMaxMessageSize* parameter points to how large a message (in bytes) can be written to the mailslot. The *lpNextSize* parameter points to the size in bytes of the next message. *GetMailslotInfo* might return the value *MAILSLOT_NO_MESSAGE*, indicating that no message is currently waiting to be received on the mailslot. A server can potentially use this parameter to poll the mailslot for incoming data, preventing your application from blocking on a *ReadFile* function call. Polling for data using this mechanism is not a good programming approach. Your application will continuously use the computer's CPU to check for incoming data—even when no messages are being processed, resulting in a slower overall performance by the computer. If you want to prevent the *ReadFile* function from blocking, we recommend using Win32 overlapped I/O. The *lpMesssageCount* parameter points to a buffer that receives the total number of messages waiting to be read. You can use this parameter for polling purposes too. The *lpReadTimeout* parameter points to a buffer that returns the amount of time in milliseconds that a read operation can wait for a message to be written to the mailslot before a timeout occurs.

The *SetMailslotInfo* API function sets the timeout values on a mailslot for how long read operations wait for incoming messages. Thus the application has the ability to change the read behavior from blocking to nonblocking mode or vice versa. *SetMailslotInfo* is defined as

```
BOOL SetMailslotInfo(
    HANDLE hMailslot,
    DWORD lReadTimeout
);
```

The *hMailslot* parameter identifies a mailslot that is returned from the *CreateMailslot* API call. The *lReadTimeout* parameter specifies the amount of time in milliseconds that a read operation can wait for a message to be written to the mailslot before a timeout occurs. If you specify 0, read operations will return immediately if no message is present. If you specify *MAILSLOT_WAIT_FOREVER*, read operations will wait forever.

Platform and Performance Considerations

Mailslots on Windows 95 and Windows 98 have three limitations that you should be aware of: 8.3-character name limits, inability to cancel blocking I/O requests, and timeout memory leaks.

8.3-Character Name Limits

Windows 95 and Windows 98 silently limit mailslot names to an 8.3-character name format. This causes interoperability problems between Windows 95 or Windows 98 and Windows NT or Windows 2000. For example, if you create or open a mailslot with the name \\.\Mailslot\Mymailslot, Windows 95 will actually create and reference the mailslot as \\.\Mailslot\Mymailsl. The *CreateMailslot* and *CreateFile* functions succeed even though name truncation occurs. If a message is sent from Windows 2000 to Windows 95 or vice versa, the message will not be received because the mailslot names do not match. If both the client and the server are running on Windows 95 machines, there isn't a problem—the name is truncated on both the client and the server. An easy way to prevent interoperability problems is to limit mailslot names to eight characters or less.

Inability to Cancel Blocking I/O Requests

Windows 95 and Windows 98 also have a problem with canceling blocking I/O requests. Mailslot servers use the *ReadFile* function to receive data. If a mailslot is created with the *MAILSLOT_WAIT_FOREVER* flag, read requests block indefinitely until data is available. If a server application is terminated when there is an outstanding *ReadFile* request, the application hangs forever. The only way to cancel the application is to reboot Windows. A possible solution is to have the server open a handle to its own mailslot in a separate thread and send data to break the blocking read request. Figure 3-3 demonstrates this solution in detail.

Figure 3-3. *Revised mailslot server*

```
// Server2.cpp

#include <windows.h>
#include <stdio.h>
#include <conio.h>

BOOL StopProcessing;

DWORD WINAPI ServeMailslot(LPVOID lpParameter);
void SendMessageToMailslot(void);

void main(void) {

    DWORD ThreadId;
    HANDLE MailslotThread;

    StopProcessing = FALSE;
    MailslotThread = CreateThread(NULL, 0, ServeMailslot, NULL,
        0, &ThreadId);

    printf("Press a key to stop the server\n");
    _getch();

    // Mark the StopProcessing flag to TRUE so that when ReadFile
    // breaks, our server thread will end
```

```
    StopProcessing = TRUE;

    // Send a message to our mailslot to break the ReadFile call
    // in our server

    SendMessageToMailslot();

    // Wait for our server thread to complete
    if (WaitForSingleObject(MailslotThread, INFINITE) == WAIT_FAILED)
    {
        printf("WaitForSingleObject failed with error %d\n",
            GetLastError());
        return;
    }
}

//
// Function: ServeMailslot
//
// Description:
//     This function is the mailslot server worker function to
//     process all incoming mailslot I/O
//
DWORD WINAPI ServeMailslot(LPVOID lpParameter)
{
    char buffer[2048];
    DWORD NumberOfBytesRead;
    DWORD Ret;
    HANDLE Mailslot;

    if ((Mailslot = CreateMailslot("\\\\.\\mailslot\\myslot", 2048,
        MAILSLOT_WAIT_FOREVER, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("Failed to create a MailSlot %d\n", GetLastError());
        return 0;
    }

    while((Ret = ReadFile(Mailslot, buffer, 2048,
        &NumberOfBytesRead, NULL)) != 0)
    {
        if (StopProcessing)
            break;

        printf("Received %d bytes\n", NumberOfBytesRead);
    }

    CloseHandle(Mailslot);

    return 0;
}

//
// Function: SendMessageToMailslot
```

```
//
// Description:
//     The SendMessageToMailslot function is designed to send a
//     simple message to our server so we can break the blocking
//     ReadFile API call
//
void SendMessageToMailslot(void)
{
    HANDLE Mailslot;
    DWORD BytesWritten;

    if ((Mailslot = CreateFile("\\\\.\\mailslot\\myslot",
        GENERIC_WRITE, FILE_SHARE_READ, NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n", GetLastError());
        return;
    }

    if (WriteFile(Mailslot, "STOP", 4, &BytesWritten, NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastError());
        return;
    }

    CloseHandle(Mailslot);
}
```

Timeout Memory Leaks

The final problem with Windows 95 and 98 worth mentioning is memory leaks, which can occur when you're using timeout values on mailslots. When you create a mailslot using the *CreateMailslot* function with a timeout value greater than 0, the *ReadFile* function leaks memory when the timeout expires and the function returns *FALSE*. After many calls to the *ReadFile* function, the system becomes unstable and subsequent *ReadFile* calls whose timers expire start returning *TRUE*. As a result, the system is no longer able to execute other MS-DOS applications. To work around this, create the mailslot with a timeout value of either 0 or *MAILSLOT_WAIT_FOREVER*. This will prevent an application from using the timeout mechanism, which causes the actual memory leak.

The Microsoft knowledge base documents the following problems and limitations. You can access the knowledge base at *http://support.microsoft.com/support/search*. We briefly describe each issue here.

- Q139715 ReadFile Returns Wrong Error Code for Mailslots
  If a server opens a mailslot using *CreateMailslot*, specifies a timeout, and then uses *ReadFile* to receive data, the *ReadFile* fails if no data is available. *GetLastError* returns an error code of 5 (access denied).

- Q192276 *GetMailslotInfo Returns Incorrect lpNextSize Value*
  If you call the API function *GetMailslotInfo* under Windows 95 OEM Service Release 2 (OSR2) or Windows 98 without a network client component installed, you receive an incorrect value (usually in the millions) or a negative number for the *lpNextSize* parameter. If you repeatedly call the function, it usually returns the correct value.

- Q170581 *Mailslot Created on Win95 Allows Only 4093 Bytes*
  If you call the *WriteFile* API function to write more than 4093 bytes to a mailslot that has been created on a Windows 95 workstation, it fails.

- Q131493 *CreateFile and Mailslots*
  The documentation for the *CreateFile* API function incorrectly describes the possible values that *CreateFile* returns when opening a client end of a mailslot.

Conclusion

This chapter introduced the mailslot networking technology, which provides an application with simple one-way interprocess data communication using the Windows redirector. One of the most useful features of mailslots is that they allow you to broadcast a message to one or more computers over a network. However, because of the broadcast capability, mailslots do not provide reliable data transmission. If you want reliable data communication using the Windows redirector, consider using named pipes—the focus of our next chapter.

Chapter 4

Named Pipes

Named pipes are a simple interprocess communication (IPC) mechanism included in Microsoft Windows NT, Windows 2000, Windows 95, and Windows 98 (but not Windows CE). Named pipes provide reliable one-way and two-way data communications among processes on the same computer or among processes on different computers across a network. Developing applications using named pipes is actually quite simple and requires no formal knowledge of underlying network transport protocols (such as TCP/IP or IPX). This is because named pipes use the Microsoft Network Provider (MSNP) redirector to form communication among processes over a network, thus hiding network protocol details from the application. One of the best reasons for using named pipes as a networking communication solution is that they take advantage of security features built into Windows NT and Windows 2000.

One possible scenario for using named pipes is developing a data management system that allows only a select group of people to perform transactions. Imagine an office setting in which you have a computer that contains company secrets. You need to have these secrets accessed and maintained by management personnel only. Let's say every employee can see the computer on the network from his or her workstation. However, you do not want regular employees to obtain access to the confidential records. Named pipes work well in this situation because you can develop a server application that, based on requests from clients, safely performs transactions on the company secrets. The server can easily limit client access to management personnel by using security features of Windows NT or Windows 2000.

What's important to remember when using named pipes as a network programming solution is that they feature a simple client/server data-communication architecture that reliably transmits data. This chapter explains how to develop named pipe client and server applications. We will start by explaining named pipe naming conventions, followed by basic pipe types. We'll then show how to implement a basic server application followed by advanced server programming details. Next we'll discuss how to develop a basic client application. By the chapter's end, we'll uncover the known problems and limitations of named pipes.

Named Pipe Implementation Details

Named pipes are designed around the Windows file system using the Named Pipe File System (NPFS) interface. As a result, client and server applications use standard Win32 file system API functions such as *ReadFile* and *WriteFile* to send and receive data. Using these API functions allows applications to take advantage of Win32 file system naming conventions and Windows NT or Windows 2000 file system security. NPFS relies on the MSNP redirector to send and receive named pipe data over a network. This makes the interface protocol-independent: when developing an application that uses named pipes to form communications among processes across a network, a programmer does not have to worry about the details of underlying network transport protocols, such as TCP and IPX. Named pipes are identified to NPFS using the Universal Naming Convention (UNC). Chapter 2 describes the UNC, the Windows redirector, and security in greater detail.

Named Pipe Naming Conventions

Named pipes are identified using the following UNC format:

```
\\server\Pipe\[path]name
```

The string above is divided into three parts: \\server, \Pipe, and \[path]name. The first string part, \\server, represents the server name in which a named pipe is created and the server that listens for incoming connections. The second part, \Pipe, is a hardcoded mandatory string requirement for identifying that this filename belongs to NPFS. The third part, \[path]name, allows applications to uniquely define and identify a named pipe name, and it can have multiple levels of directories. For example, the following name types are legal for identifying a named pipe:

```
\\myserver\PIPE\mypipe
\\Testserver\pipe\cooldirectory\funtest\jim
\\.\Pipe\Easynamedpipe
```

The server string portion can be represented as a dot (.) or a server name.

Byte Mode and Message Mode

Named pipes offer two basic communication modes: byte mode and message mode. In byte mode, messages travel as a continuous stream of bytes between the client and the server. This means that a client application and a server application do not know precisely how many bytes are being read from or written to a pipe at any given moment. So a write on one side will not always result in a same-size read on the other. This allows a client and a server to transfer data without regard to the contents of the data. In message mode, the client and the server send and receive data in discrete units. Every time a message is sent on the pipe, it must be read as a complete message. Figure 4-1 compares the two pipe modes.

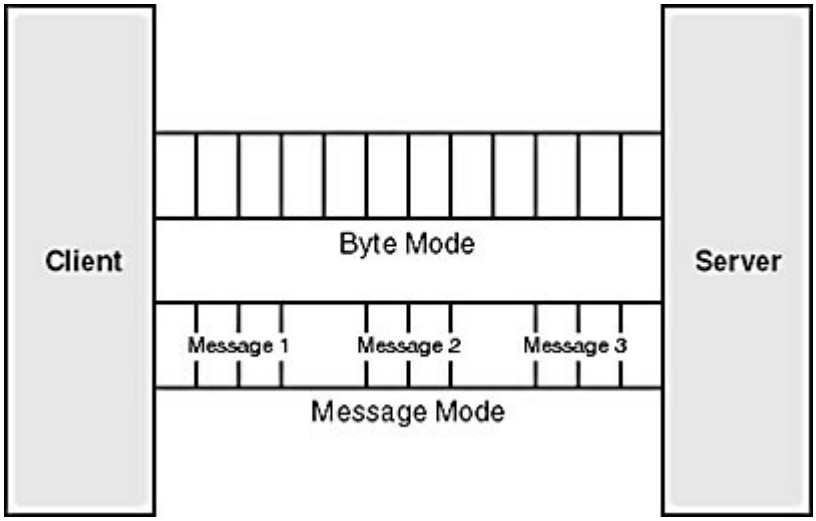

Figure 4-1. *Byte mode and message mode*

Compiling Applications

When you build a named pipe client or server application using Microsoft Visual C++, your application must include the Winbase.h file in your program files. But if your application includes Windows.h—as most do—you can omit Winbase.h. Your application is also responsible for linking with Kernel32.lib, which typically is configured with the Visual C++ linker flags.

Error Codes

All Win32 API functions (except *CreateFile* and *CreateNamedPipe*) that are used in developing named pipe client and server applications return the value 0 when they fail. *CreateFile* and *CreateNamedPipe* return *INVALID_HANDLE_VALUE*. When either of these functions fails, applications should call the *GetLastError* function to retrieve specific information about the failure. For a complete list of error codes, consult the header file Winerror.h or see Appendix C, "Winsock Error Codes."

Basic Server and Client

Named pipes feature a simple client/server design architecture in which data can flow in both a unidirectional and a bidirectional manner between a client and server. This is useful because it allows you to send and receive data whether your application is a client or a server. The main difference between a named pipe server and a client application is that a named pipe server is the only process capable of creating a named pipe and accepting pipe client connections. A client application is capable only of connecting to an existing named pipe server. Once a connection is formed between a client application and a server application, both processes are capable of reading and writing data on a pipe using standard Win32 functions such as *ReadFile* and *WriteFile* . Note that a named pipe server application can operate only on Windows NT or Windows 2000—Windows 95 and Windows 98 do not permit applications to create a named pipe. This limitation makes it impossible to form communications directly between two Windows 95 or Windows 98 computers. However, Windows 95 and Windows 98 clients can form connections to Windows NT and Windows 2000 computers.

Server Details

Implementing a named pipe server requires developing an application to create one or more named pipe instances, which can be accessed by clients. To a server, a pipe instance is nothing more than a handle used to accept a connection from a local or remote client application. The following steps describe how to write a basic server application:

1. Create a named pipe instance handle using the CreateNamedPipe API function.
2. Use the *ConnectNamedPipe* API function to listen for a client connection on the named pipe instance.
3. Receive data from and send data to the client using the *ReadFile* and *WriteFile* API functions.
4. Close down the named pipe connection using the *DisconnectNamedPipe* API function.
5. Close the named pipe instance handle using the *CloseHandle* API function.

First, your server process needs to create a named pipe instance using the *CreateNamedPipe* API call, which is defined as

```
HANDLE CreateNamedPipe(
    LPCTSTR lpName,
    DWORD dwOpenMode,
    DWORD dwPipeMode,
    DWORD nMaxInstances,
    DWORD nOutBufferSize,
    DWORD nInBufferSize,
    DWORD nDefaultTimeOut,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

The first parameter, *lpName* , specifies the name of a named pipe. The name must have the following UNC form:

```
\\.\Pipe\[path]name
```

Notice that the server name is represented as a dot, which represents the local machine. You cannot create a named pipe on a remote computer. The *[path]name* part of the parameter must represent a unique name. This might simply be a filename, or it might be a full directory path followed by a filename.

The *dwOpenMode* parameter describes the directional, I/O control, and security modes of a pipe when it is

created. Table 4-1 describes all the available flags that can be used. A pipe can be created using a combination of these flags by ORing them together.

The *PIPE_ACCESS_* flags determine flow direction on a pipe between a client and a server. A pipe can be opened as bidirectional (two-way) using the *PIPE_ACCESS_DUPLEX* flag: data can flow in both directions between the client and the server. In addition, you can also control the direction of data flow by opening the pipe as unidirectional (one-way) using the flag *PIPE_ACCESS_INBOUND* or *PIPE_ACCESS_OUTBOUND* : data can flow only one way from the client to the server or vice versa. Figure 4-2 describes the flag combinations further and shows the flow of data between a client and a server.

Table 4-1. *Named pipe open mode flags*

Open Mode	*Flags*	*Description*
Directional		
	PIPE_ACCESS_DUPLEX	The pipe is bidirectional: both the server and client processes can read from and write data to the pipe.
	PIPE_ACCESS_OUTBOUND	The flow of data in the pipe goes from server to client only.
	PIPE_ACCESS_INBOUND	The flow of data in the pipe goes from client to server only.
I/O Control		
	FILE_FLAG_WRITE_THROUGH	Works only for byte-mode pipes. Functions writing to a named pipe do not return until the data written is transmitted across the network and is in the pipe's buffer on the remote computer.
	FILE_FLAG_OVERLAPPED	Allows functions that perform read, write, and connect operations to use overlapped I/O.
Security		
	WRITE_DAC	Allows your application to have write access to the named pipe's DACL.
	ACCESS_SYSTEM_SECURITY	Allows your application to have write access to the named pipe's SACL.
	WRITE_OWNER	Allows your application to have write access to the named pipe's owner and group SID.

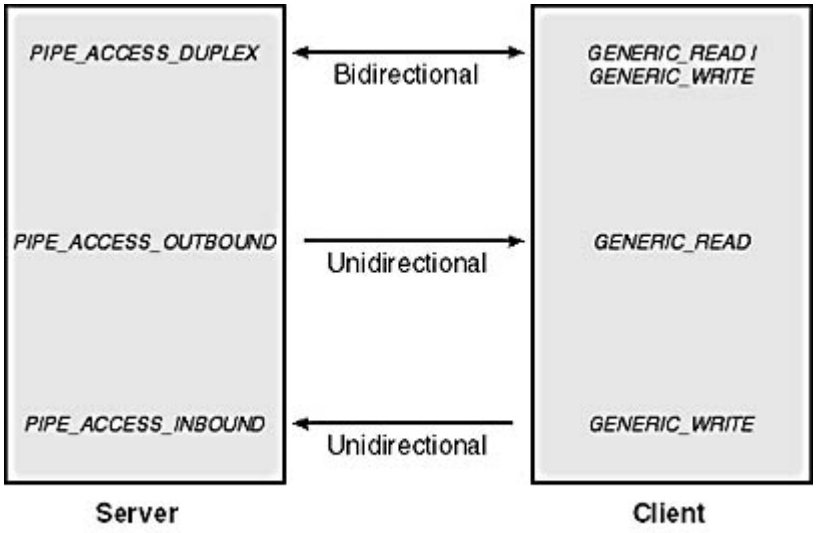

Figure 4-2. *Mode flags and flow direction*

The next set of *dwOpenMode* flags controls I/O behavior on a named pipe from the server's perspective. The *FILE_FLAG_WRITE_THROUGH* flag controls the write operations so that functions writing to a named pipe do not return until the data written is transmitted across the network and is in the pipe's buffer on the remote computer. This flag works only for byte-mode named pipes when the client and the server are on different computers. The *FILE_FLAG_OVERLAPPED* flag allows functions performing read, write, and connect operations to return immediately, even if those functions take significant time to complete. We will discuss the details of overlapped I/O when we develop an advanced server later in this chapter.

The last set of *dwOpenMode* flags described in Table 4-1 controls the server's ability to access the security descriptor that is created by a named pipe. If your application needs to modify or update the pipe's security descriptor after the pipe is created, you should set these flags accordingly to permit access. The *WRITE_DAC* flag allows your application to update the pipe's discretionary access control list (DACL), whereas *ACCESS_SYSTEM_SECURITY* allows access to the pipe's system access control list (SACL). The *WRITE_OWNER* flag allows you to change the pipe's owner and group security ID (SID). For example, if you want to deny access to a particular user who has access rights to your pipe, you can modify the pipe's DACL using security API functions. Chapter 2 discusses DACLs, SACLs, and SIDs in greater detail.

CreateNamedPipe 's *dwPipeMode* parameter specifies the read, write, and wait operating modes of a pipe. Table 4-2 describes all the available mode flags that can be used. The flags can be issued by ORing one flag from each mode category. If a pipe is opened as byte-oriented using the *PIPE_READMODE_BYTE* | *PIPE_TYPE_BYTE* mode flags, data can be read and written only as a stream of bytes. This means that when you read and write data to a pipe, you do not have to balance each read and write because your data does not have any message boundaries. For example, if a sender writes 500 bytes to a pipe, a receiver might want to read 100 bytes at a time until it receives all of the data. To establish clear boundaries around messages, place the pipe in message-oriented mode using the flags *PIPE_READMODE_MESSAGE* | *PIPE_TYPE_MESSAGE* , meaning each read and write must be balanced. For example, if a sender writes a 500-byte message to a pipe, the receiver must provide the *ReadFile* function a 500-byte or larger buffer when reading data. If the receiver fails to do so, *ReadFile* will fail with error *ERROR_MORE_DATA* . You can also combine *PIPE_TYPE_MESSAGE* with *PIPE_READMODE_BYTE* , allowing a sender to write messages to a pipe and the receiver to read an arbitrary amount of bytes at a time. The message delimiters will be ignored in the data stream. You cannot mix the *PIPE_TYPE_BYTE* flag with the *PIPE_READMODE_MESSAGE* flag. Doing so will cause the *CreateNamedPipe* function to fail with error *ERROR_INVALID_PARAMETER* because no message delimiters are in the I/O stream when data is written into the pipe as bytes. The *PIPE_WAIT* or *PIPE_NOWAIT* flag can also be combined with read and write mode flags. The *PIPE_WAIT* flag places a pipe in blocking mode and the *PIPE_NOWAIT* flag places a pipe in nonblocking mode. In blocking mode, I/O operations such as *ReadFile* will block until the I/O request is complete. This is the default behavior if you do not specify any flags. The nonblocking mode flag *PIPE_NOWAIT* is designed to allow I/O operations to return immediately. However, it should not be used to achieve asynchronous I/O in Win32 applications. It is included to provide backward compatibility with older Microsoft LAN Manager 2.0 applications. The *ReadFile* and *WriteFile* functions allow applications to accomplish asynchronous I/O using Win32 overlapped I/O, which will be demonstrated later in this chapter.

Table 4-2. *Named pipe read/write mode flags*

Mode	Flags	Description
Write		
	PIPE_TYPE_BYTE	Data is written to the pipe as a stream of bytes.
	PIPE_TYPE_MESSAGE	Data is written to the pipe as a stream of messages.
Read		
	PIPE_READMODE_BYTE	Data is read from the pipe as a stream of bytes.
	PIPE_READMODE_MESSAGE	Data is read from the pipe as a stream of messages.
Wait		
	PIPE_WAIT	Blocking mode is enabled.
	PIPE_NOWAIT	Nonblocking mode is enabled.

> NOTE
>
> The *PIPE_NOWAIT* flag is obsolete and should not be used in Win32 environments to accomplish asynchronous I/O. It is included in this book to provide backward compatibility with older Microsoft LAN Manager 2.0 software.

The *nMaxInstances* parameter specifies how many instances or pipe handles can be created for a named pipe. A pipe instance is a connection from a local or remote client application to a server application that created the

Named Pipe. Acceptable values are in the range 1 through *PIPE_UNLIMITED_INSTANCES* . For example, if you want to develop a server that can service only five client connections at a time, set this parameter to 5. If you set this parameter to *PIPE_UNLIMITED_INSTANCES* , the number of pipe instances that can be created is limited only by the availability of system resources.

CreateNamedPipe 's *nOutBufferSize* and *nInBufferSize* parameters represent the number of bytes to reserve for internal input and output buffer sizes. These sizes are advisory in that every time a named pipe instance is created, the system sets up inbound and/or outbound buffers using the nonpaged pool (the physical memory used by the operating system). The buffer size specified should be reasonable (not too large) so that your system will not run out of nonpaged pool memory, but it should also be large enough to accommodate typical I/O requests. If an application attempts to write data that is larger than the buffer sizes specified, the system will try to automatically expand the buffers to accommodate the data using nonpaged pool memory. For practical purposes, applications should size these internal buffers to match the size of the application's send and receive buffers used when calling *ReadFile* and *WriteFile* .

The *nDefaultTimeOut* parameter specifies the default timeout value (how long a client will wait to connect to a named pipe) in milliseconds. This affects only client applications that use the *WaitNamedPipe* function to determine when an instance of a named pipe is available to accept connections. We will discuss this concept in greater detail later in this chapter, when we develop a named pipe client application.

The *lpSecurityAttributes* parameter allows the application to specify a security descriptor for a named pipe and determines whether a child process can inherit the newly created handle. If this parameter is specified as *NULL* , the named pipe gets a default security descriptor and the handle cannot be inherited. A default security descriptor grants the named pipe the same security limits and access controls as the process that created it following the Windows NT and Windows 2000 security model described in Chapter 2 . An application can apply access control restrictions to a pipe by setting access privileges for particular users and groups in a *SECURITY_DESCRIPTOR* structure using security API functions. If a server wants to open access to any client, you should assign a null discretionary access control list (DACL) to the *SECURITY_DESCRIPTOR* structure.

After you successfully receive a handle from *CreateNamedPipe* , which is known as a pipe instance, you have to wait for a connection from a named pipe client. This connection can be made through the *ConnectNamedPipe* API function, which is defined as

```
BOOL ConnectNamedPipe(
    HANDLE hNamedPipe,
    LPOVERLAPPED lpOverlapped
);
```

The *hNamedPipe* parameter represents the pipe instance handle returned from *CreateNamedPipe* . The *lpOverlapped* parameter allows this API function to operate asynchronously, or in nonblocking mode, if the pipe was created using the *FILE_FLAG_OVERLAPPED* flag, which is known as Win32 overlapped I/O. If this parameter is specified as *NULL* , *ConnectNamedPipe* blocks until a client forms a connection to the server. We will discuss overlapped I/O in greater detail when you learn to create a more advanced named pipe server later in this chapter.

Once a named pipe client successfully connects to your server, the *ConnectNamedPipe* API call completes. The server is then free to send data to a client using the *WriteFile* API function and to receive data from the client using *ReadFile* . Once the server has finished communicating with a client, it should call *DisconnectNamedPipe* to close the communication session. Figure 4-3 demonstrates how to write a simple server application that can communicate with one client.

Figure 4-3. *Simple named pipe server*

```
// Server.cpp

#include <windows.h>
#include <stdio.h>

void main(void)
{
```

```
    HANDLE PipeHandle;
    DWORD BytesRead;
    CHAR buffer[256];

    if ((PipeHandle = CreateNamedPipe("\\\\.\\Pipe\\Jim",
        PIPE_ACCESS_DUPLEX, PIPE_TYPE_BYTE | PIPE_READMODE_BYTE, 1,e
        0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateNamedPipe failed with error %d\n",
            GetLastError());
        return;
    }

    printf("Server is now running\n");

    if (ConnectNamedPipe(PipeHandle, NULL) == 0)
    {
        printf("ConnectNamedPipe failed with error %d\n",
            GetLastError());
        CloseHandle(PipeHandle);
        return;
    }

    if (ReadFile(PipeHandle, buffer, sizeof(buffer),
        &BytesRead,  NULL) <= 0)
    {
        printf("ReadFile failed with error %d\n", GetLastError());
        CloseHandle(PipeHandle);
        return;
    }

    printf("%.*s\n", BytesRead, buffer);

    if (DisconnectNamedPipe(PipeHandle) == 0)
    {
        printf("DisconnectNamedPipe failed with error %d\n",
            GetLastError());
        return;
    }

    CloseHandle(PipeHandle);
}
```

Building Null Discretionary Access Control Lists (Null DACLs)

When applications create securable objects such as files and named pipes on Windows NT or Windows 2000 using Win32 API functions, the operating system grants the applications the ability to set up access control rights by specifying a *SECURITY_ATTRIBUTES* structure, defined as:

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD   nLength;
    LPVOID  lpSecurityDescriptor;
    BOOL    bInheritHandle
} SECURITY_ATTRIBUTES;
```

The *lpSecurityDescriptor* field defines the access rights for an object in a *SECURITY_DESCRIPTOR* structure. A *SECURITY_DESCRIPTOR* structure contains a DACL field that defines which users and groups can access the object. If you set this field to *NULL* , any user or group can access your resource.

Applications cannot directly access a *SECURITY_DESCRIPTOR* structure and must use Win32 security API functions to do so. If you want to assign a null DACL to a *SECURITY_DESCRIPTOR* structure, you must do the following:

1. Create and initialize a *SECURITY_DESCRIPTOR* structure by calling the *InitializeSecurityDescriptor* API function.
2. Assign a null DACL to the *SECURITY_DESCRIPTOR* structure by calling the *SetSecurityDescriptorDacl* API function.

After you successfully build a new *SECURITY_DESCRIPTOR* structure, you must assign it to the *SECURITY_ATTRIBUTES* structure. Now you are ready to begin calling Win32 functions such as *CreateNamedPipe* with your new *SECURITY_ATTRIBUTES* structure, which contains a null DACL. The following code fragment demonstrates how to call the security API functions needed to accomplish this.

```
// Create new SECURITY_ATTRIBUTES and SECURITY_DESCRIPTOR
// structure objects
SECURITY_ATTRIBUTES sa;
SECURITY_DESCRIPTOR sd;

// Initialize the new SECURITY_DESCRIPTOR object to empty values
if (InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION)
    == 0)
{
    printf("InitializeSecurityDescriptor failed with error %d\n",
        GetLastError());
    return;
}

// Set the DACL field in the SECURITY_DESCRIPTOR object to NULL
if (SetSecurityDescriptorDacl(&sd, TRUE, NULL, FALSE) == 0)
{
    printf("SetSecurityDescriptorDacl failed with error %d\n",
        GetLastError());
    return;
}

// Assign the new SECURITY_DESCRIPTOR object to the
// SECURITY_ATTRIBUTES object
sa.nLength = sizeof(SECURITY_ATTRIBUTES);
sa.lpSecurityDescriptor = &sd;
sa.bInheritHandle = TRUE;
```

Advanced Server

Figure 4-3 demonstrates how to develop a named pipe server application that handles only a single pipe instance. All of the API calls operate in a synchronous mode in which each call waits until an I/O request is complete. A named pipe server is also capable of having multiple pipe instances so that clients can form two or more connections to the server; the number of pipe instances is limited by the number specified in the *nMaxInstances* parameter of the *CreateNamedPipe* API call. To handle more than one pipe instance, a server must consider using multiple threads or asynchronous Win32 I/O mechanisms—such as overlapped I/O and completion ports—to service each pipe instance. Asynchronous I/O mechanisms allow a server to service all pipe instances simultaneously from a single application thread. Our discussion will demonstrate how to develop advanced servers using threads and overlapped I/O. See Chapter 8 for more information on completion ports as they apply to Windows sockets.

Threads

Developing an advanced server that can support more than one pipe instance using threads is simple. All you need to do is create one thread for each pipe instance and service each instance using the techniques we described earlier for the simple server. Figure 4-4 demonstrates a server that is capable of serving five pipe instances. The application is an echo server that reads data from a client and echoes the data back.

Figure 4-4. *Advanced named pipe server using threads in Win32*

```
// Threads.cpp

#include <windows.h>
#include <stdio.h>
#include <conio.h>

#define NUM_PIPES 5

DWORD WINAPI PipeInstanceProc(LPVOID lpParameter);

void main(void)
{
    HANDLE ThreadHandle;
    INT i;
    DWORD ThreadId;

    for(i = 0; i < NUM_PIPES; i++)
    {
        // Create a thread to serve each pipe instance
        if ((ThreadHandle = CreateThread(NULL, 0, PipeInstanceProc,
            NULL, 0, &ThreadId)) == NULL)
        {
            printf("CreateThread failed with error %\n",
                GetLastError());
            return;
        }
        CloseHandle(ThreadHandle);
    }

    printf("Press a key to stop the server\n");
```

```
    _getch();
}

//
// Function: PipeInstanceProc
//
// Description:
//     This function handles the communication details of a single
//     named pipe instance
//
DWORD WINAPI PipeInstanceProc(LPVOID lpParameter)
{
    HANDLE PipeHandle;
    DWORD BytesRead;
    DWORD BytesWritten;
    CHAR Buffer[256];

    if ((PipeHandle = CreateNamedPipe("\\\\.\\PIPE\\jim",
        PIPE_ACCESS_DUPLEX, PIPE_TYPE_BYTE | PIPE_READMODE_BYTE,
        NUM_PIPES, 0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateNamedPipe failed with error %d\n",
            GetLastError());
        return 0;
    }

    // Serve client connections forever
    while(1)
    {
        if (ConnectNamedPipe(PipeHandle, NULL) == 0)
        {
            printf("ConnectNamedPipe failed with error %d\n",
                GetLastError());
            break;
        }

        // Read data from and echo data to the client until
        // the client is ready to stop
        while(ReadFile(PipeHandle, Buffer, sizeof(Buffer),
            &BytesRead,  NULL) > 0)
        {
            printf("Echo %d bytes to client\n", BytesRead);

            if (WriteFile(PipeHandle, Buffer, BytesRead,
                &BytesWritten, NULL) == 0)
            {
                printf("WriteFile failed with error %d\n",
                    GetLastError());
                break;
            }
        }

        if (DisconnectNamedPipe(PipeHandle) == 0)
```

```
        {
            printf("DisconnectNamedPipe failed with error %d\n",
                GetLastError());
            break;
        }
    }

    CloseHandle(PipeHandle);
    return 0;
}
```

To develop your server to handle five pipe instances, start by calling the *CreateThread* API function. *CreateThread* starts five execution threads, all of which execute the *PipeInstanceProc* function simultaneously. The *PipeInstanceProc* function operates exactly like the basic server application (Figure 4-3) except that it reuses a named pipe handle by calling the *DisconnectNamedPipe* API function, which closes a client's session to the server. Once an application calls *DisconnectNamedPipe* , it is free to service another client by calling the *ConnectNamedPipe* function with the same pipe instance handle.

Overlapped I/O

Overlapped I/O is a mechanism that allows Win32 API functions such as *ReadFile* and *WriteFile* to operate asynchronously when I/O requests are made. This is accomplished by passing an *OVERLAPPED* structure to these API functions and later retrieving the results of an I/O request through the original *OVERLAPPED* structure using the *GetOverlappedResult* API function. When a Win32 API function is invoked with an overlapped structure, the call returns immediately.

To develop an advanced named pipe server that can manage more than one named pipe instance using overlapped I/O, you need to call *CreateNamedPipe* with the *nMaxInstances* parameter set to a value greater than 1. You also must set the *dwOpenMode* flag to *FILE_FLAG_OVERLAPPED* . Figure 4-5 demonstrates how to develop this advanced named pipe server. The application is an echo server that reads data from a client and writes the data back.

Figure 4-5. *Advanced named pipe server using Win32 overlapped I/O*

```
// Overlap.cpp

#include <windows.h>
#include <stdio.h>

#define NUM_PIPES 5
#define BUFFER_SIZE 256

void main(void)
{
    HANDLE PipeHandles[NUM_PIPES];
    DWORD BytesTransferred;
    CHAR Buffer[NUM_PIPES][BUFFER_SIZE];
    INT i;
    OVERLAPPED Ovlap[NUM_PIPES];
    HANDLE Event[NUM_PIPES];

    // For each pipe handle instance, the code must maintain the
    // pipes' current state, which determines if a ReadFile or
    // WriteFile is posted on the named pipe. This is done using
    // the DataRead variable array. By knowing each pipe's
```

```
// current state, the code can determine what the next I/O
// operation should be.
BOOL DataRead[NUM_PIPES];

DWORD Ret;
DWORD Pipe;

for(i = 0; i < NUM_PIPES; i++)
{
    // Create a named pipe instance
    if ((PipeHandles[i] = CreateNamedPipe("\\\\.\\PIPE\\jim",
        PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED,
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE, NUM_PIPES,
        0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateNamedPipe for pipe %d failed "
            "with error %d\n", i, GetLastError());
        return;
    }

    // Create an event handle for each pipe instance. This
    // will be used to monitor overlapped I/O activity on
    // each pipe.
    if ((Event[i] = CreateEvent(NULL, TRUE, FALSE, NULL))
        == NULL)
    {
        printf("CreateEvent for pipe %d failed with error %d\n",
            i, GetLastError());
        continue;
    }

    // Maintain a state flag for each pipe to determine when data
    // is to be read from or written to the pipe
    DataRead[i] = FALSE;

    ZeroMemory(&Ovlap[i], sizeof(OVERLAPPED));
    Ovlap[i].hEvent = Event[i];

    // Listen for client connections using ConnectNamedPipe()
    if (ConnectNamedPipe(PipeHandles[i], &Ovlap[i]) == 0)
    {
        if (GetLastError() != ERROR_IO_PENDING)
        {
            printf("ConnectNamedPipe for pipe %d failed with"
                " error %d\n", i, GetLastError());
            CloseHandle(PipeHandles[i]);
            return;
        }
    }
}

printf("Server is now running\n");
```

```
// Read and echo data back to Named Pipe clients forever
while(1)
{
    if ((Ret = WaitForMultipleObjects(NUM_PIPES, Event,
        FALSE, INFINITE)) == WAIT_FAILED)
    {
        printf("WaitForMultipleObjects failed with error %d\n",
            GetLastError());
        return;
    }

    Pipe = Ret - WAIT_OBJECT_0;

    ResetEvent(Event[Pipe]);

    // Check overlapped results, and if they fail, reestablish
    // communication for a new client; otherwise, process read
    // and write operations with the client

    if (GetOverlappedResult(PipeHandles[Pipe], &Ovlap[Pipe],
        &BytesTransferred, TRUE) == 0)
    {
        printf("GetOverlapped result failed %d start over\n",
            GetLastError());

        if (DisconnectNamedPipe(PipeHandles[Pipe]) == 0)
        {
            printf("DisconnectNamedPipe failed with error %d\n",
                GetLastError());
            return;
        }

        if (ConnectNamedPipe(PipeHandles[Pipe],
            &Ovlap[Pipe]) == 0)
        {
            if (GetLastError() != ERROR_IO_PENDING)
            {
                // Severe error on pipe. Close this
                // handle forever.
                printf("ConnectNamedPipe for pipe %d failed with"
                    " error %d\n", i, GetLastError());
                CloseHandle(PipeHandles[Pipe]);
            }
        }

        DataRead[Pipe] = FALSE;
    }
    else
    {
        // Check the state of the pipe. If DataRead equals
        // FALSE, post a read on the pipe for incoming data.
        // If DataRead equals TRUE, then prepare to echo data
```

```
            // back to the client.

            if (DataRead[Pipe] == FALSE)
            {
                // Prepare to read data from a client by posting a
                // ReadFile operation

                ZeroMemory(&Ovlap[Pipe], sizeof(OVERLAPPED));
                Ovlap[Pipe].hEvent = Event[Pipe];

                if (ReadFile(PipeHandles[Pipe], Buffer[Pipe],
                    BUFFER_SIZE, NULL, &Ovlap[Pipe]) == 0)
                {
                    if (GetLastError() != ERROR_IO_PENDING)
                    {
                        printf("ReadFile failed with error %d\n",
                        GetLastError());
                    }
                }

                DataRead[Pipe] = TRUE;
            }
            else
            {
                // Write received data back to the client by
                // posting a WriteFile operation
                printf("Received %d bytes, echo bytes back\n",
                    BytesTransferred);

                ZeroMemory(&Ovlap[Pipe], sizeof(OVERLAPPED));
                Ovlap[Pipe].hEvent = Event[Pipe];

                if (WriteFile(PipeHandles[Pipe], Buffer[Pipe],
                    BytesTransferred, NULL, &Ovlap[Pipe]) == 0)
                {
                    if (GetLastError() != ERROR_IO_PENDING)
                    {
                        printf("WriteFile failed with error %d\n",
                        GetLastError());
                    }
                }

                DataRead[Pipe] = FALSE;
            }
        }
    }
}
```

For the server application to service five pipe instances at a time, it must call *CreateNamedPipe* five times to retrieve an instance handle for each pipe. After the server retrieves all the instance handles, it begins to listen for clients by calling *ConnectNamedPipe* asynchronously five times using an overlapped I/O structure for each pipe. As clients form connections to the server, all I/O is processed asynchronously. When clients disconnect, the server reuses each pipe instance handle by calling *DisconnectNamedPipe* and reissuing a *ConnectNamedPipe* call.

Security impersonation

One of the best reasons for using named pipes as a network programming solution is that they rely on Windows NT and Windows 2000 security features to control access when clients attempt to form communication to a server. Windows NT and Windows 2000 security offers security impersonation, which allows a named pipe server application to execute in the security context of a client. When a named pipe server executes, it normally operates at the security context permission level of the process that starts the application. For example, if a person with administrator privileges starts up a named pipe server, the server has the ability to access almost every resource on a Windows NT or Windows 2000 system. Such security access for a named pipe server is bad if the *SECURITY_DESCRIPTOR* structure specified in *CreateNamedPipe* allows all users to access your named pipe.

When a server accepts a client connection using the *ConnectNamedPipe* function, it can make its execution thread operate in the security context of the client by calling the *ImpersonateNamedPipeClient* API function, which is defined as

```
BOOL ImpersonateNamedPipeClient(
    HANDLE hNamedPipe
);
```

The *hNamedPipe* parameter represents the pipe instance handle that is returned from *CreateNamedPipe* . When this function is called, the operating system changes the thread security context of the server to the security context of the client. This is quite handy: if your server is designed to access resources such as files, it will do so using the client's access rights, thereby allowing your server to preserve access control to resources regardless of who started the process.

When a server thread executes in a client's security context, it does so through a security impersonation level. There are four basic impersonation levels: Anonymous, Identification, Impersonation, and Delegation. Security impersonation levels govern the degree to which a server can act on behalf of a client. We will discuss these impersonation levels in greater detail when we develop a client application later in this chapter. After the server finishes processing a client's session, it should call *RevertToSelf* to return to its original thread execution security context. The *RevertToSelfAPI* function is defined as

```
BOOL RevertToSelf(VOID);
```

This function does not have any parameters.

Client Details

Implementing a named pipe client requires developing an application that forms a connection to a named pipe server. Clients cannot create named pipe instances. However, clients do open handles to preexisting instances from a server. The following steps describe how to write a basic client application:

1. Wait for a named pipe instance to become available using the *WaitNamedPipe* API function.
2. Connect to the named pipe using the *CreateFile* API function.
3. Send data to and receive data from the server using the *WriteFile* and *ReadFile* API functions.
4. Close the named pipe session using the *CloseHandle* API function.

Before forming a connection, clients need to check for the existence of a named pipe instance using the *WaitNamedPipe* function, which is defined as

```
BOOL WaitNamedPipe(
    LPCTSTR lpNamedPipeName,
```

```
    DWORD nTimeOut
);
```

The *lpNamedPipeName* parameter represents the named pipe you are trying to connect to. The *nTimeOut* parameter represents how long a client is willing to wait for a pipe's server process to have a pending *ConnectNamedPipe* operation on the pipe.

After *WaitNamedPipe* successfully completes, the client needs to open a handle to the server's named pipe instance using the *CreateFile* API function. *CreateFile* is defined as

```
HANDLE CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
```

The *lpFileName* parameter is the name of the pipe you are trying to open; the name must conform to the named pipe naming conventions mentioned earlier.

The *dwDesiredAccess* parameter defines the access mode and should be set to *GENERIC_READ* for reading data off the pipe and *GENERIC_WRITE* for writing data to the pipe. These flags can also be specified together by ORing both flags. The access mode must be compatible with how the pipe was created in the server. Match the mode specified in the *dwOpenMode* parameter of *CreateNamedPipe* , as described earlier. For example, if the server creates a pipe with *PIPE_ ACCESS_ INBOUND* , the client should specify *GENERIC_WRITE* .

The *dwShareMode* parameter should be set to 0 because only one client is capable of accessing a pipe instance at a time. The *lpSecurityAttributes* parameter should be set to *NULL* unless you need a child process to inherit the client's handle. This parameter is incapable of specifying security controls because *CreateFile* is not capable of creating named pipe instances. The *dwCreationDisposition* parameter should be set to *OPEN_EXISTING* , which means that the *CreateFile* function will fail if the named pipe does not exist.

The *dwFlagsAndAttributes* parameter should always be set to *FILE_ATTRIBUTE_NORMAL* . Optionally, you can specify the *FILE_FLAG_WRITE_THROUGH* , *FILE_FLAG_OVERLAPPED* , and *SECURITY_SQOS_PRESENT* flags by ORing them with the *FILE_ATTRIBUTE_NORMAL* flag. The *FILE_FLAG_WRITE_THROUGH* and *FILE_FLAG_OVERLAPPED* flags behave like the server's mode flags. The *SECURITY_SQOS_PRESENT* flag controls client impersonation security levels in a named pipe server. Security impersonation levels govern the degree to which a server process can act on behalf of a client process. A client can specify this information when it connects to a server. When the client specifies the *SECURITY_SQOS_PRESENT* flag, it must use one or more of the security flags listed below.

- *SECURITY_ANONYMOUS* Specifies to impersonate the client at the Anonymous impersonation security level. The server process cannot obtain identification information about the client, and it cannot execute in the security context of the client.
- *SECURITY_IDENTIFICATION* Specifies to impersonate the client at the Identification impersonation security level. The server process can obtain information about the client, such as security identifiers and privileges, but it cannot execute in the security context of the client. This is useful for named pipe clients that want to allow the server to identify the client but not to act as the client.
- *SECURITY_IMPERSONATION* Specifies to impersonate the client at the Impersonation security level. The client wants to allow the server process to obtain information about the client and execute in the client's security context on its local system. Using this flag, the client allows the server to access any local resource on the server as the client. The server, however, cannot impersonate the client on remote systems.
- *SECURITY_DELEGATION* Specifies to impersonate the client at the Delegation impersonation security

level. The server process can obtain information about the client and execute in the client's security context on its local system and on remote systems.

> NOTE
>
> *SECURITY_DELEGATION* works only if the server process is running on Windows 2000. Windows NT 4 does not implement security delegation.

- *SECURITY_CONTEXT_TRACKING* Specifies that the security-tracking mode is dynamic. If this flag is not specified, security-tracking mode is static.
- *SECURITY_EFFECTIVE_ONLY* Specifies that only the enabled aspects of the client's security context are available to the server. If you do not specify this flag, all aspects of the client's security context are available.

Named pipe security impersonation is described earlier in this chapter in the "Server Details " section.

The final parameter of *CreateFile* , *hTemplateFile* , does not apply to named pipes and should be specified as *NULL* . If *CreateFile* completes without an error, the client application can begin to send and receive data on the named pipe using the *ReadFile* and *WriteFile* functions. Once the application is finished processing data, it can close down the connection using the *CloseHandle* function.

The program in Figure 4-6 is a simple named pipe client that demonstrates the API calls needed to successfully develop a basic named pipe client application. When this application successfully connects to a named pipe, it writes the message, "This is a test" to the server.

Figure 4-6. *Simple named pipe client*

```
// Client.cpp

#include <windows.h>
#include <stdio.h>

#define PIPE_NAME "\\\\.\\Pipe\\jim"

void main(void)
{
    HANDLE PipeHandle;
    DWORD BytesWritten;

    if (WaitNamedPipe(PIPE_NAME, NMPWAIT_WAIT_FOREVER) == 0)
    {
        printf("WaitNamedPipe failed with error %d\n",
            GetLastError());
        return;
    }

    // Create the named pipe file handle
    if ((PipeHandle = CreateFile(PIPE_NAME,
        GENERIC_READ | GENERIC_WRITE, 0,
        (LPSECURITY_ATTRIBUTES) NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        (HANDLE) NULL)) == INVALID_HANDLE_VALUE)
```

```
    {
        printf("CreateFile failed with error %d\n", GetLastError());
        return;
    }

    if (WriteFile(PipeHandle, "This is a test", 14, &BytesWritten,
        NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastError());
        CloseHandle(PipeHandle);
        return;
    }

    printf("Wrote %d bytes", BytesWritten);

    CloseHandle(PipeHandle);
}
```

Other API Calls

There are several additional named pipe functions that we haven't touched on yet. The first set of these API functions—*CallNamedPipe* and *TransactNamedPipe*—is designed to reduce coding complexity in an application. Both functions perform a write and read operation in one call. The *CallNamedPipe* function allows a client application to connect to a message-type pipe (and waits if an instance of the pipe is not available), writes to and reads from the pipe, and then closes the pipe. This is practically an entire client application written in one call. *CallNamedPipe* is defined as

```
BOOL CallNamedPipe(
    LPCTSTR lpNamedPipeName,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesRead,
    DWORD nTimeOut
);
```

The *lpNamedPipeName* parameter is a string that represents the named pipe in UNC form. The *lpInBuffer* and *nInBufferSize* parameters represent the address and the size of the buffer that the application uses to write data to the server. The *lpOutBuffer* and *nOutBufferSize* parameters represent the address and the size of the buffer that the application uses to retrieve data from the server. The *lpBytesRead* parameter receives the number of bytes read from the pipe. The *nTimeOut* parameter specifies how many milliseconds to wait for the named pipe to be available.

The *TransactNamedPipe* function can be used in both a client and a server application. It is designed to combine read and write operations in one API call, thus optimizing network I/O by reducing send and receive transactions in the MSNP redirector. *TransactNamedPipe* is defined as

```
BOOL TransactNamedPipe(
    HANDLE hNamedPipe,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesRead,
    LPOVERLAPPED lpOverlapped
);
```

The *hNamedPipe* parameter identifies the named pipe returned by the *CreateNamedPipe* or *CreateFile* API functions. The *lpInBuffer* and *nInBufferSize* parameters represent the address and the size of the buffer that the application uses to write data to the pipe. The *lpOutBuffer* and *nOutBufferSize* parameters represent the address and the size of the buffer that the application uses to retrieve data from the pipe. The *lpBytesRead* parameter receives the number of bytes read from the pipe. The *lpOverlapped* parameter allows this *TransactNamedPipe* to operate asynchronously using overlapped I/O.

The next set of functions—*GetNamedPipeHandleState*, *SetNamedPipeHandleState*, and *GetNamedPipeInfo*—are designed to make named pipe client and server communication more flexible at run time. For example, you can use these functions to change the operating mode of a pipe at run time from message mode to byte mode and vice versa. *GetNamedPipeHandleState* retrieves information such as the operating mode (message mode and byte mode), pipe instance count, and buffer caching information about a specified named pipe. The information that

GetNamedPipeHandleState returns can vary during the lifetime of an instance of the named pipe. *GetNamedPipeHandleState* is defined as

```
BOOL GetNamedPipeHandleState(
    HANDLE hNamedPipe,
    LPDWORD lpState,
    LPDWORD lpCurInstances,
    LPDWORD lpMaxCollectionCount,
    LPDWORD lpCollectDataTimeout,
    LPTSTR lpUserName,
    DWORD nMaxUserNameSize
);
```

The *hNamedPipe* parameter identifies the named pipe returned by the *CreateNamedPipe* or *CreateFile* function. The *lpState* parameter is a pointer to a variable that receives the current operating mode of the pipe handle. The *lpState* parameter can return the value *PIPE_NOWAIT* or the value *PIPE_READMODE_MESSAGE*. The *lpCurInstances* parameter is a pointer to a variable that receives the number of current pipe instances. The *lpMaxCollectionCount* parameter receives the maximum number of bytes to be collected on the client's computer before transmission to the server. The *lpCollectDataTimeout* parameter receives the maximum time in milliseconds that can pass before a remote named pipe transfers information over a network. The *lpUserName* and *nMaxUserNameSize* parameters represent a buffer that receives a null-terminated string containing the user name string of the client application.

The *SetNamedPipeHandleState* function allows you to change the pipe characteristics retrieved with *GetNamedPipeHandleState*. *SetNamedPipeHandleState* is defined as

```
BOOL SetNamedPipeHandleState(
    HANDLE hNamedPipe,
    LPDWORD lpMode,
    LPDWORD lpMaxCollectionCount,
    LPDWORD lpCollectDataTimeout
);
```

The *hNamedPipe* parameter identifies the named pipe returned by *CreateNamedPipe* or *CreateFile*. The *lpMode* parameter sets the operating mode of a pipe. The *lpMaxCollectionCount* parameter specifies the maximum number of bytes collected on the client computer before data is transmitted to the server. The *lpCollectDataTimeout* parameter specifies the maximum time in milliseconds that can pass before a remote named pipe client transfers information over the network.

The *GetNamedPipeInfo* API function is used to retrieve buffer size and maximum pipe instance information. *GetNamedPipeInfo* is defined as

```
BOOL GetNamedPipeInfo(
    HANDLE hNamedPipe,
    LPDWORD lpFlags,
    LPDWORD lpOutBufferSize,
    LPDWORD lpInBufferSize,
    LPDWORD lpMaxInstances
);
```

The *hNamedPipe* parameter identifies the named pipe returned by *CreateNamedPipe* or *CreateFile*. The *lpFlags* parameter retrieves the type of the named pipe and determines whether it is a server or a client and whether the pipe is in byte mode or message mode. The *lpOutBufferSize* parameter determines the size in bytes of the internal buffer for outgoing data. The *lpInBufferSize* parameter receives the size in bytes of the internal buffer for incoming data. The *lpMaxInstance* parameter receives the maximum number of pipe instances that can be created.

The final API function, *PeekNamedPipe*, allows an application to look at the data in a named pipe without removing it from the pipe's internal buffer. This function is useful if an application wants to poll for incoming data to avoid blocking on the *ReadFile* API call. The function can also be useful for applications that need to examine data before they actually receive it. For example, an application might want to adjust its application buffers based on the size of incoming messages. *PeekNamedPipe* is defined as

```
BOOL PeekNamedPipe(
   HANDLE hNamedPipe,
   LPVOID lpBuffer,
   DWORD nBufferSize,
   LPDWORD lpBytesRead,
   LPDWORD lpTotalBytesAvail,
   LPDWORD lpBytesLeftThisMessage
);
```

The *hNamedPipe* parameter identifies the named pipe returned by *CreateNamedPipe* or *CreateFile*. The *lpBuffer* and *nBufferSize* parameters represent the receiving buffer along with the receiving buffer size to retrieve data from the pipe. The *lpBytesRead* parameter receives the number of bytes read from the pipe into the *lpBuffer* parameter. The *lpTotalBytesAvail* parameter receives the total number of bytes that are available to be read from the pipe. The *lpBytesLeftThisMessage* parameter receives the number of bytes remaining in a message if a pipe is opened in message mode. If a message cannot fit in the *lpBuffer* parameter, the remaining bytes in a message are returned. This parameter always returns 0 for byte-mode named pipes.

Platform and Performance Considerations

The Microsoft Knowledge Base documents the following problems and limitations. You can access the Knowledge Base at *http://support.microsoft.com/support*. The following are brief descriptions of each issue.

- Q100291 *Restriction on Named-Pipe Names*

  If a pipe named \\.\Pipe\Mypipes is created, it is not possible to subsequently create a pipe named \\.\Pipe\Mypipes\Pipe1, because \\.\Pipe\Mypipes is already a pipe name and cannot be used as a subdirectory.

- Q119218 *Named Pipe Write Limited to 64K*

  The *WriteFile* API function returns *FALSE* and *GetLastError* returns *ERROR_MORE_DATA* when *WriteFile* writes to a message-mode named pipe using a buffer greater than 64 KB.

- Q110148 ERROR_INVALID_PARAMETER *from* WriteFile *or* ReadFile

  The *WriteFile* or *ReadFile* function call can fail with the error *ERROR_INVALID_PARAMETER* if you are operating on a named pipe and using overlapped I/O. A possible cause for the failure is that the *Offset* and *OffsetHigh* members of the *OVERLAPPED* structure are not set to 0.

- Q180222 WaitNamedPipe *and Error 253 in Windows 95*

  In Windows 95, when *WaitNamedPipe* fails because of an invalid pipe name passed as the first parameter, *GetLastError* returns error 253, which is not listed as a possible error code for this function. When you run the same code on Windows NT 4, the error code 161 (*ERROR_BAD_PATHNAME*) appears. To work around the problem, resolve error 253 the same way as error 161, *ERROR_BAD_PATHNAME*.

- Q141709 *Limit of 49 Named Pipe Connections from a Single Workstation*

  If a named pipe server creates more than 49 distinctly named pipes, a single client on a remote computer cannot connect more than 49 pipes on the named pipe server.

- Q126645 *Access Denied When Opening a Named Pipe from a Service*

  If a service running in the Local System account attempts to open a named pipe on a computer running Windows NT, the operation can fail with an Access Denied error (error 5).

Conclusion

This chapter introduced the named pipe networking technology, which provides a simple client/server data-communication architecture that reliably transmits data. The interface relies on the Windows redirector to transmit data over a network. A major benefit of named pipes is that it takes advantage of Windows NT and Windows 2000 security features—an advantage offered by no other networking technology described in this book. This chapter concludes Part I, our discussion of file system network programming using the Windows redirector. Part II discusses the Winsock technology that allows applications to communicate directly over a network transport protocol.

Part II

The Winsock API

Part II of this book is dedicated to Winsock programming on Win32 platforms. Winsock is the preferred interface for accessing a variety of underlying network protocols and is available in varying forms on every Win32 platform. Winsock is a network programming interface and not a protocol. The Winsock interface inherits a great deal from the Berkeley (BSD) Sockets implementation on UNIX platforms, which is capable of accessing multiple networking protocols. In Win32 environments, the interface has finally evolved into a truly protocol-independent interface, especially with the release of Winsock 2.

The next three chapters describe protocol and Winsock basics, including addressing for each protocol and a simple Winsock client/server sample. The later chapters describe new features in Winsock 2, such as transport service providers, name space providers, and Quality of Service (QOS). What might be confusing about some of these technologies is that they are in the Winsock 2 specification and Winsock 2 is supported on all the current Win32 platforms (except Windows CE, which we will discuss in this section), but not all these features are implemented on the given platforms. These limitations are pointed out wherever applicable. This section assumes you have basic knowledge of Winsock (or BSD sockets) and are somewhat familiar with basic client/server Winsock terminology.

Chapter 5

Network Principles and Protocols

The main motivation for creating the Winsock 2 specification was to provide a protocol-independent transport interface. This is wonderful in that it provides one familiar interface for network programming over a variety of network transports, but you should still be aware of the network protocol characteristics. This chapter covers the traits that you should be aware of when utilizing a particular protocol, including some basic networking principles. Additionally, we'll discuss how to programmatically query Winsock for protocol information, and we'll examine the basic steps necessary to create a socket for a given protocol.

Protocol Characteristics

The first part of this chapter discusses the basic characteristics found in the world's available transport protocols. This information is meant to provide a bit of background on the type of behavior that protocols adhere to as well as to inform you, the programmer, of how a protocol will behave in an application.

Message-Oriented

A protocol is said to be message-oriented if for each discrete write command it transmits those and only those bytes as a single message on the network. This also means that when the receiver requests data, the data returned is a discrete message written by the sender. The receiver will not get more than one message. In Figure 5-1, for example, the workstation on the left submits messages of 128, 64, and 32 bytes destined for the workstation on the right. The receiving workstation issues three read commands with a 256-byte buffer. Each call in succession returns 128, 64, and 32 bytes. The first call to read does not return all three packets, even if all the packets have been received. This logic is known as preserving message boundaries and is often desired when structured data is exchanged. A network game is a good example of preserving message boundaries. Each player sends all other players a packet with positional information. The code behind such communication is simple: one player requests a packet of data, and that player gets exactly one packet of positional information from another player in the game.

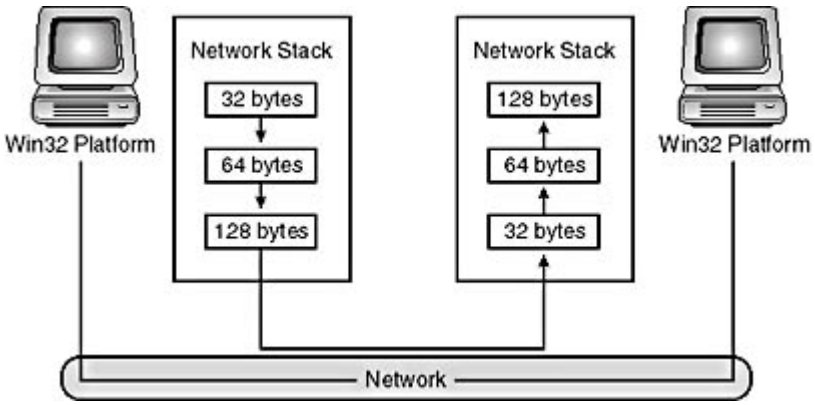

Figure 5-1. *Datagram services*

A protocol that does not preserve these message boundaries is often referred to as a stream-based protocol. Be aware that the term "stream-based" is often loosely applied to imply additional characteristics. Stream service is defined as the transmitting of data in a continual process: the receiver reads as much data as is available with no respect to message boundaries. For the sender, this means that the system is allowed to break up the original message into pieces or lump several messages together to form a bigger packet of data. On the receiving end, the network stack reads data as it arrives and buffers it for the process. When the process requests an amount of data, the system returns as much data as possible without overflowing the buffer supplied by the client call. In Figure 5-2, the sender submits packets of 128, 64, and 32 bytes; however, the local system stack is free to gather the data into larger packets. In this case, the second two packets are transmitted together. The decision to lump discrete packets of data together is based on a number of factors, such as the maximum transmit unit or the Nagle algorithm. In TCP/IP, the Nagle algorithm consists of a host waiting for data to accumulate before sending it on the wire. The host will wait until it has a large enough chunk of data to send or until a predetermined amount of time elapses. When implementing the Nagle algorithm the host's peer waits a predetermined amount of time for outgoing data before sending an acknowledgement to the host so that the peer doesn't have to send a packet with only the acknowledgement. Sending many packets of a small size is inefficient and adds substantial overhead for error checking and acknowledgments.

On the receiving end, the network stack pools together all incoming data for the given process. Take a look at Figure 5-2. If the receiver performs a read with a 256-byte buffer, all 224 bytes are returned at once. If the receiver requests that only 20 bytes be read, the system returns only 20 bytes.

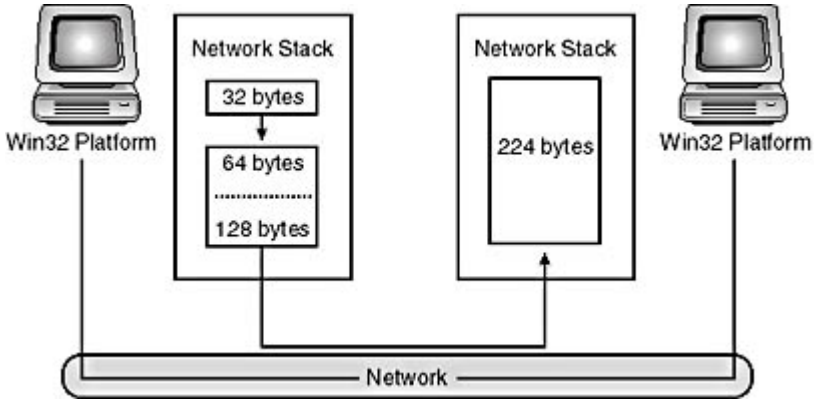

Figure 5-2. *Stream services*

Pseudo-stream

Pseudo-stream is a term often applied to a system with a message-based protocol that sends data in discrete packets, which the receiver reads and buffers in a pool so that the receiving application reads data chunks of any size. Combining the sender in Figure 5-1 with the receiver in Figure 5-2 illustrates how pseudo-streams work. The sender must send each individual packet separately, but the receiver is free to gather them. For the most part, treat pseudo-streaming as you would a normal stream-oriented protocol.

Connection-Oriented and Connectionless

A protocol usually provides either connection-oriented services or connectionless services. In connection-oriented services, a path is established between the two communicating parties before any data is exchanged. This ensures that there is a route between the two parties in addition to ensuring that both parties are alive and responding. This also means that establishing a communication channel between two participants requires a great deal of overhead. Additionally, most connection-oriented protocols guarantee delivery, further increasing overhead as additional computations are performed to verify correctness. On the other hand, a connectionless protocol makes no guarantees that the recipient is listening. A connectionless service is similar to the postal service: the sender addresses a letter to a particular person and puts it in the mail. The sender doesn't know whether the recipient is expecting to receive a letter or whether severe storms are preventing the post office from delivering the message.

Reliability and Ordering

Reliability and ordering are perhaps the most critical characteristics to be aware of when designing an application to use a particular protocol. In most cases, reliability and ordering are closely tied to whether a protocol is connectionless or connection-oriented. Reliability, or guaranteed delivery, ensures that each byte of data from the sender will reach the intended recipient unaltered. An unreliable protocol does not ensure that each byte arrives, and it makes no guarantee as to the data's integrity.

Ordering has to do with the order in which the data arrives at the recipient. A protocol that preserves ordering ensures that the recipient receives the data in the exact order it was sent. Obviously, a protocol that does not preserve order makes no such guarantees.

Reliability and ordering are closely tied to whether a protocol is connectionless or connection-oriented. In the case of connection-oriented communications, if you are already making the extra effort to establish a clear communication channel between the two participants, you usually want to guarantee data integrity and data ordering. In most cases, connection-oriented protocols do guarantee reliability. The next section discusses actual protocols and their characteristics. Note that by assuring packet ordering, you do not automatically guarantee data integrity. Of course, the great benefit of connectionless protocols is their speed: they don't bother to establish a virtual connection to the recipient. Why slow this down with error checking? This is why connectionless protocols generally don't guarantee data integrity or ordering, while connection-oriented protocols do. Why would anyone use datagrams with all these faults? Generally, connectionless protocols are an order of magnitude faster than connection-oriented communications. No checks need to be made for factors such as data integrity and acknowledgments of received data, factors that add a great deal of complexity to sending even small amounts of data. Datagrams are useful for noncritical data transfers. Datagrams are well suited for applications like the game

example that we discussed earlier: each player can use datagrams to periodically send his or her positions within the game to every other player. If one client misses a packet, it quickly receives another, giving the player an appearance of seamless communication.

Graceful Close

A graceful close is associated only with connection-oriented protocols. In a graceful close, one side initiates the shutting down of a communication session and the other side still has the opportunity to read pending data on the wire or the network stack. A connection-oriented protocol that does not support graceful closes causes an immediate termination of the connection and loss of any data not read by the receiving end whenever either side closes the communication channel. In the case of TCP, each side of a connection has to perform a close to fully terminate the connection. The initiating side sends a segment (datagram) with a FIN control flag to the peer. Upon receipt, the peer sends an ACK control flag back to the initiating side to acknowledge receipt of the FIN, but the peer is still able to send more data. The FIN control flag signifies that no more data will be sent from the side originating the close. Once the peer decides it no longer needs to send data, it too issues a FIN, which the initiator acknowledges with an ACK control flag. At this point, the connection has been completely closed.

Broadcast Data

To broadcast data is to be able to send data from one workstation so that all other workstations on the local area network can receive it. This feature is available to connectionless protocols because all machines on the LAN can pick up and process a broadcast message. The drawback to using broadcast messages is that every machine has to process the message. For example, let's say the user broadcasts a message on the LAN, and the network card on each machine picks up the message and passes it up to the network stack. The stack then cycles through all network applications to see whether they should receive this message. Usually a majority of the machines on the network are not interested and simply discard the data. However, each machine still has to spend time processing the packet to see whether any applications are interested in it. Consequently, high-broadcast traffic can bog down machines on a LAN as each workstation inspects the packet. In general, routers do not propagate broadcast packets.

Multicast Data

Multicasting is the ability of one process to send data that will be received by one or more recipients. The method by which a process joins a multicast session differs depending on the underlying protocol. For example, under the IP protocol, multicasting is a modified form of broadcasting. IP multicasting requires that all hosts interested in sending or receiving data join a special group. When a process wishes to join a multicast group, a filter is added on the network card so that data bound to only that group address will be picked up by the network hardware and propagated up the network stack to the appropriate process. Video conferencing applications often use multicasting. Chapter 11 covers multicast programming from Winsock as well as other critical multicasting issues.

Quality of Service

Quality of service (QOS) is an application's ability to request certain bandwidth requirements to be dedicated for exclusive use. One good use for quality of service is real-time video streaming. In order for the receiving end of a real-time video streaming application to display a smooth, clear picture, the data being sent must fall within certain restrictions. In the past, an application would buffer data and play back frames from the buffer to maintain a smooth video. If there is a period during which data is not being received fast enough, the playback routine has a certain number of buffered frames that it can play. QOS allows bandwidth to be reserved on the network, so frames can be read off the wire within a set of guaranteed constraints. Theoretically this means that the same real-time video streaming application can use QOS and eliminate the need to buffer any frames. QOS is discussed in detail in Chapter 12.

Partial Messages

Partial messages apply only to message-oriented protocols. Let's say an application wants to receive a message but the local computer has received only part of the data. This can be a common occurrence, especially if the sending computer is transmitting large messages. The local machine might not have enough resources free to contain the whole message. In reality, most message-oriented protocols impose a reasonable limit on the

maximum size of a datagram, so this particular event is not encountered often. However, most datagram protocols support messages of a size large enough to require being broken up into a number of smaller chunks for transmission on the physical medium. Thus the possibility exists that when a user's application requests to read a message, the user's system might have received only a portion of the message. If the protocol supports partial messages, the reader is notified that the data being returned is only a part of the whole message. If the protocol does not support partial messages, the underlying network stack holds onto the pieces until the whole message arrives. If for some reason the whole message does not arrive, most unreliable protocols that lack support for partial messages will simply discard the incomplete datagram.

Routing Considerations

One important consideration is whether a protocol is routable. If a protocol is routable, a successful communication path can be set up (either a virtual connection-oriented circuit or a data path for datagram communication) between two workstations, no matter what network hardware lies between them. For example, machine A is on a separate subnet from machine B. A router linking the two subnets separates the two machines. A routable protocol realizes that machine B is not on the same subnet as machine A; therefore, it directs the packet to the router, which decides how to best forward the data so that it reaches machine B. A nonroutable protocol is not able to make such provisions: the router drops any packets of nonroutable protocols that it receives. The router does not forward a packet from a nonroutable protocol even if the packet's intended destination is on the connected subnet. NetBEUI is the only protocol supported by Win32 platforms that is not capable of being routed.

Other Characteristics

Each protocol supported on Win32 platforms presents characteristics that are specialized or unique. Also, a myriad of other protocol characteristics, such as byte ordering and maximum transmission size, can be used to describe every protocol available on networks today. Not all of those characteristics are necessarily critical to writing a successful Winsock application. Winsock 2 provides a facility to enumerate each available protocol provider and query its characteristics. The third section of this chapter explains this function and presents a code sample.

Supported Protocols

One of the most useful features offered by Win32 platforms is the capability to simultaneously support many different network protocols. As you read in Chapter 2, the Windows redirector ensures that network requests are routed to the right protocols and subsystems; however, with Winsock you can write network applications that directly utilize any one of these protocols. Chapter 6 discusses how machines on a network are addressed using the different protocols available to a workstation. One of the benefits of using the Winsock programming interface is that it is a protocol-independent interface. A majority of all operations are common no matter which protocol is used. However, you must understand how workstations are addressed in order to locate and connect to a peer for network communication.

Supported Win32 Network Protocols

Win32 platforms support a wide variety of protocols. Each protocol is usually capable of multiple behaviors. For example, Internet Protocol (IP) is capable of both connection-oriented stream services and connectionless datagram services. Table 5-1 provides most of the various protocols available and some of the behaviors each supports.

Table 5-1. *The characteristics of available protocols*

Protocol	Name	Message Type	Connection Type	Reliable	Packet Ordering	Graceful Close	Supports Broadcast	Supports Multipoint	Quality of Service	Maximum Message Size (Bytes)
IP	MSAFD TCP	Stream	Connection	Yes	Yes	Yes	No	No	No	None
	MSAFD UDP	Message	Connectionless	No	No	No	Yes	Yes	No	65467
	RSVP TCP	Stream	Connection	Yes	Yes	Yes	No	No	Yes	None
	RSVP UDP	Message	Connectionless	No	No	No	Yes	Yes	Yes	65467
IPX/SPX	MSAFD nwlnkipx [IPX]	Message	Connectionless	No	No	No	Yes	Yes	No	576
	MSAFD nwlnkspx [SPX]	Message	Connection	Yes	Yes	No	No	No	No	None
	MSAFD nwlnkspx [SPX] [Pseudo-stream]	Message	Connection	Yes	Yes	No	No	No	No	None
	MSAFD nwlnkspx [SPXII]	Message	Connection	Yes	Yes	Yes	No	No	No	None
	MSAFD nwlnkspx [SPXII] [Pseudo-stream]	Message	Connection	Yes	Yes	Yes	No	No	No	None
NetBIOS	Sequential Packets	Message	Connection	Yes	Yes	No	No	No	No	64 KB (65535)
	Datagrams	Message	Connectionless	No	No	No	Yes* [SP25]	No	No	64 KB (65535)
AppleTalk	MSAFD AppleTalk [ADSP]	Message	Connection	Yes	Yes	Yes	No	No	No	64 KB (65535)
	MSAFD AppleTalk [ADSP] [Pseudo-stream]	Message	Connection	Yes	Yes	Yes	No	No	No	None
	MSAFD AppleTalk [PAP]	Message	Connection	Yes	Yes	Yes	No	No	No	4096
	MSAFD AppleTalk [RTMP]	Stream	Connectionless	No	No	No	No	No	No	None
	MSAFD AppleTalk [ZIP]	Stream	Connectionless	No	No	No	No	No	No	None
ATM	MSAFD ATM AAL5	Stream	Connection	No	Yes	No	No	Yes	Yes	None
	Native ATM (AAL5)	Message	Connection	No	Yes	No	No	Yes	Yes	None
Infrared Sockets	MSAFD Irda [IrDA]	Stream	Connection	Yes	Yes	Yes	No	No	No	None

* NetBIOS supports datagrams sent to either unique or group NetBIOS clients. It does not support blanket broadcasts.

Windows CE Network Protocols

Windows CE differs from the other Win32 platforms in that it supports only TCP/IP as a network transport protocol.

Additionally, Windows CE supports only the Winsock 1.1 specification, which means that most of the new Winsock 2 features covered in this section don't apply to this platform. Windows CE does support NetBIOS over TCP/IP through a redirector, but it does not offer any kind of programming interface to NetBIOS through the native NetBIOS API or through Winsock.

Winsock 2 Protocol Information

Winsock 2 provides a method for determining which protocols are installed on a given workstation and returning a variety of characteristics for each protocol. If a protocol is capable of multiple behaviors, each distinct behavior type has its own catalog entry within the system. For example, if you install TCP/IP on your system, there will be two IP entries: one for TCP, which is reliable and connection-oriented, and one for UDP, which is unreliable and connectionless.

The function call to obtain information on installed network protocols is *WSAEnumProtocols* and is defined as

```
int WSAEnumProtocols (
    LPINT lpiProtocols,
    LPWSAPROTOCOL_INFO lpProtocolBuffer,
    LPDWORD lpdwBufferLength
);
```

This function supersedes the Winsock 1.1 function *EnumProtocols*, which is the necessary function for Windows CE. The only difference is that *WSAEnumProtocols* returns an array of *WSAPROTOCOL_INFO* structures, whereas *EnumProtocols* returns an array of *PROTOCOL_INFO* structures that contain fewer fields than the *WSAPROTOCOL_INFO* structure (but more or less the same information). The *WSAPROTOCOL_INFO* structure is defined as

```
typedef struct _WSAPROTOCOL_INFOW {
    DWORD             dwServiceFlags1;
    DWORD             dwServiceFlags2;
    DWORD             dwServiceFlags3;
    DWORD             dwServiceFlags4;
    DWORD             dwProviderFlags;
    GUID              ProviderId;
    DWORD             dwCatalogEntryId;
    WSAPROTOCOLCHAIN  ProtocolChain;
    int               iVersion;
    int               iAddressFamily;
    int               iMaxSockAddr;
    int               iMinSockAddr;
    int               iSocketType;
    int               iProtocol;
    int               iProtocolMaxOffset;
    int               iNetworkByteOrder;
    int               iSecurityScheme;
    DWORD             dwMessageSize;
    DWORD             dwProviderReserved;
    WCHAR             szProtocol[WSAPROTOCOL_LEN + 1];
} WSAPROTOCOL_INFOW, FAR * LPWSAPROTOCOL_INFOW;
```

Before you can call a Winsock function, you must load the correct version of the Winsock library. The Winsock initialization routine is *WSAStartup*, defined as

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);
```

The first parameter is the version of the Winsock library that you want to load. For current Win32 platforms, the latest Winsock 2 library is version 2.2. The only exception is Windows CE, which supports only Winsock version 1.1. If you wanted Winsock version 2.2, you could either specify the value (0x0202) or use the macro *MAKEWORD(2, 2)*. The high-order byte specifies the minor version number, while the low-order byte specifies the major version number.

The second parameter is a structure, *WSADATA*, that is returned upon completion. *WSADATA* contains information about the version of Winsock that *WSAStartup* loaded. Table 5-2 lists the individual fields of the *WSADATA* structure, which is actually defined as

```
typedef struct WSAData {
    WORD            wVersion;
    WORD            wHighVersion;
    char            szDescription[WSADESCRIPTION_LEN + 1];
    char            szSystemStatus[WSASYS_STATUS_LEN + 1];
    unsigned short  iMaxSockets;
    unsigned short  iMaxUdpDg;
    char FAR *      lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

For the most part, the only useful information returned in the *WSADATA* structure is *wVersion* and *wHighVersion*. The entries pertaining to maximum sockets and maximum UDP size should be obtained from the catalog entry for the specific protocol you are using. The above section on *WSAEnumProtocols* discusses this.

Table 5-2. *Member fields of the WSADATA structure*

Field	Description
wVersion	The Winsock version the caller is expected to use
wHighVersion	The highest Winsock version supported by the loaded library, usually the same value as *wVersion*
szDescription	A text description of the loaded library
szSystemStatus	A text string containing relevant status or configuration information
iMaxSockets	Maximum number of sockets (ignore this field for Winsock 2 or later)
iMaxUdpDg	Maximum UDP datagram size (ignore this field for Winsock 2 or later)
lpVendorInfo	Vendor-specific information (ignore this field for Winsock 2 or later)

When you are finished with the Winsock library and no longer want to call any Winsock functions, the companion routine *WSACleanup* unloads the library and frees any resources. This function is defined as

```
int WSACleanup (void);
```

Keep in mind that for each call to *WSAStartup* a matching call to *WSACleanup* will be needed because each startup call increments the reference count to the loaded Winsock DLL, requiring an equal number of calls to *WSACleanup* to decrement the count.

Note that Winsock 2 is fully compatible with all Winsock 1.1 function calls. Thus an application written to the Winsock 1.1 specification will be able to run if it loads the Winsock 2 library, as the Winsock 1.1 functions are mapped through their Winsock 2 equivalents.

The easiest way to call *WSAEnumProtocols* is to make the first call with *lpProtocolBuffer* equal to *NULL* and *lpdwBufferLength* set to 0. The call fails with *WSAENOBUFS*, but *lpdwBufferLength* then contains the correct size of the buffer required to return all the protocol information. Once you allocate the correct buffer size and make another call with the supplied buffer, the function returns the number of *WSAPROTOCOL_INFO* structures returned. At this point, you can step through the structures to find the protocol entry with your required attributes. The sample program Enum.c on the companion CD-ROM enumerates all installed protocols and prints out the characteristics of each protocol.

The most commonly used field of the *WSAPROTOCOL_INFO* structure is *dwServiceFlags1*, which is a bit field for the various protocol attributes. Table 5-3 lists the various bit flags that can be set in the field and describes the meaning of each property. To check for the presence of a particular property, select the appropriate property flag and perform a bitwise AND of the property and the *dwServiceFlags1* field. If the resultant value is nonzero, that property is present in the given protocol; otherwise, it isn't.

Table 5-3. *Protocol flags*

Property	*Meaning*
XP1_CONNECTIONLESS	This protocol provides connectionless service. If not set, the protocol supports connection-oriented data transfers.
XP1_GUARANTEED_DELIVERY	This protocol guarantees that all data sent will reach the intended recipient.
XP1_GUARANTEED_ORDER	This protocol guarantees that the data will arrive in the order in which it was sent and that it will not be duplicated. However, this does not guarantee delivery.
XP1_MESSAGE_ORIENTED	This protocol honors message boundaries.
XP1_PSEUDO_STREAM	This protocol is message-oriented, but the message boundaries are ignored on the receiver side.
XP1_GRACEFUL_CLOSE	This protocol supports two-phase closes: each party is notified of the other's intent to close the communication channel. If not set, only abortive closes are performed.
XP1_EXPEDITED_DATA	This protocol supports urgent data (out-of-band data).
XP1_CONNECT_DATA	This protocol supports transferring data with the connection request.
XP1_DISCONNECT_DATA	This protocol supports transferring data with the disconnect request.
XP1_SUPPORT_BROADCAST	This protocol supports the broadcast mechanism.
XP1_SUPPORT_MULTIPOINT	This protocol supports multipoint or multicast mechanisms.
XP1_MULTIPOINT_CONTROL_PLANE	If this flag is set, the control plane is rooted. Otherwise, it is nonrooted.
XP1_MULTIPOINT_DATA_PLANE	If this flag is set, the data plane is rooted. Otherwise, it is nonrooted.

XP1_QOS_SUPPORTED	This protocol supports QOS requests.
XP1_UNI_SEND	This protocol is unidirectional in the send direction.
XP1_UNI_RECV	This protocol is unidirectional in the receive direction.
XP1_IFS_HANDLES	The socket descriptors returned by the provider are Installable File System (IFS) handles and can be used in API functions such as *ReadFile* and *WriteFile*.
XP1_PARTIAL_MESSAGE	The *MSG_PARTIAL* flag is supported in *WSASend* and *WSASendTo*.

Most of these flags will be discussed in one or more of the following chapters, so we won't go into detail about the full meaning of each flag now. The other fields of importance are *iProtocol*, *iSocketType*, and *iAddressFamily*. The *iProtocol* field defines which protocol this entry belongs to. The *iSocketType* field is important if the protocol is capable of multiple behaviors, such as stream-oriented connections or datagram connections. Finally, *iAddressFamily* is used to distinguish the correct addressing structure to use for the given protocol. These three entries are of great importance when creating a socket for a given protocol and will be discussed in detail in the next section.

Windows Sockets

Now that you are familiar with the various protocols available and their attributes, we'll take a look at using these protocols from Winsock. If you're familiar with Winsock, you know that the API is based on the concept of a socket. A socket is a handle to a transport provider. In Win32, a socket is not the same thing as a file descriptor and therefore is a separate type, *SOCKET*. Two functions create a socket:

```
SOCKET WSASocket (
    int af,
    int type,
    int protocol,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    GROUP g,
    DWORD dwFlags
);

SOCKET socket (
    int af,
    int type,
    int protocol
);
```

The first parameter, *af*, is the address family of the protocol. For example, if you want to create either a UDP or a TCP socket, use the constant *AF_INET* to indicate the Internet Protocol (IP). The second parameter, *type*, is the socket type of the protocol. A socket type can be one of five values: *SOCK_STREAM*, *SOCK_DGRAM*, *SOCK_SEQPACKET*, *SOCK_RAW*, and *SOCK_RDM*. The third parameter is *protocol*. This field is used to qualify a specific transport if there are multiple entries for the given address family and socket type. Table 5-4 shows the values used for the address family, socket type, and protocol fields for a given network transport.

Table 5-4. *Socket parameters*

Protocol	*Address Family*	*Socket Type*		*Protocol*
Internet Protocol (IP)	*AF_INET*	TCP	*SOCK_STREAM*	*IPPROTO_IP*
		UDP	*SOCK_DGRAM*	*IPPROTO_UDP*
		Raw sockets	*SOCK_RAW*	*IPPROTO_RAW* *IPPROTO_ICMP*
IPX/SPX	*AF_NS*	MSAFD nwlnkipx [IPX]	*SOCK_DGRAM*	*NSPROTO_IPX*
	AF_IPX	MSAFD nwlnkspx [SPX]	*SOCK_SEQPACKET*	*NSPROTO_SPX*
		MSAFD nwlnkspx [SPX] [Pseudo-stream]	*SOCK_STREAM*	*NSPROTO_SPX*
		MSAFD nwlnkspx [SPXII]	*SOCK_SEQPACKET*	*NSPROTO_SPXII*

		MSAFD nwlnkspx [SPXII] [Pseudo-stream]	*SOCK_STREAM*	*NSPROTO_SPXII*
NetBIOS	*AF_NETBIOS*	Sequential Packets	*SOCK_SEQPACKET*	LANA number
		Datagrams	*SOCK_DGRAM*	LANA number
AppleTalk	*AF_APPLETALK*	MSAFD AppleTalk [ADSP]	*SOCK_RDM*	*ATPROTO_ADSP*
		MSAFD AppleTalk [ADSP] [Pseudo-stream]	*SOCK_STREAM*	*ATPROTO_ADSP*
		MSAFD AppleTalk [PAP]	*SOCK_RDM*	*ATPROTO_PAP*
		MSAFD AppleTalk [RTMP]	*SOCK_DGRAM*	*DDPPROTO_RTMP*
		MSAFD AppleTalk [ZIP]	*SOCK_DGRAM*	*DDPPROTO_ZIP*
ATM	*AF_ATM*	MSAFD ATM AAL5 Native ATM (AAL5)	*SOCK_RAW ATMPROTO_AAL5*	*SOCK_RAW ATMPROTO_AAL5*
Infrared Sockets	*AF_IRDA*	MSAFD Irda [IrDA]	*_SOCK_STREAM*	*SOCK_STREAM IRDA_PROTO*

The first three parameters for creating a socket are organized in three tiers. The first and most important parameter is the address family. This specifies which protocol is being used. It also dictates the valid options for the second and third parameters. For example, choosing the ATM address family (*AF_ATM*) limits you to only raw sockets (*SOCK_RAW*) for the socket type. Likewise, by selecting an address family and a socket type, you are limited as to the protocol you choose. However, it is possible to pass a 0 for the *protocol* parameter. The system then chooses a transport provider based on the other two parameters, *af* and *type*. When enumerating the catalog entries for protocols, check the *dwProviderFlags* entry of the *WSAPROTOCOL_INFO* structure. If this field is set to *PFL_MATCHES_PROTOCOL_ZERO*, this is the default transport that will be used if the protocol parameter to *socket* or *WSASocket* is 0.

If you are using the *WSASocket* function and have enumerated all protocols using *WSAEnumProtocols*, you can select a *WSAPROTOCOL_INFO* structure and pass that to the *WSASocket* as the *lpProtocolInfo* parameter. If you then specify the constant *FROM_PROTOCOL_INFO* in all of the first three parameters (*af*, *type*, and *protocol*), the values from the *WSAPROTOCOL_INFO* structure you supplied are used instead. This is how you specify an exact protocol entry.

The last two flags of *WSASocket* are simple. The group parameter is always 0 because no version of Winsock supports socket groups. The *dwFlags* parameter is used to specify one or more of the following flags:

- *WSA_FLAG_OVERLAPPED*
- *WSA_FLAG_MULTIPOINT_C_ROOT*
- *WSA_FLAG_MULTIPOINT_C_LEAF*
- *WSA_FLAG_MULTIPOINT_D_ROOT*
- *WSA_FLAG_MULTIPOINT_D_LEAF*

The first flag, *WSA_FLAG_OVERLAPPED*, is used to specify that this socket is capable of overlapped I/O, which is

one of the possible communication models available in Winsock. This topic is covered in detail in Chapter 8. If you create a socket using the *socket* call, *WSA_FLAG_OVERLAPPED* is set by default. In general, it is a good idea to always set this flag when using *WSASocket*. The last four flags deal with multicast sockets.

Raw Sockets

When creating a socket with *WSASocket*, you can pass a *WSAPROTOCOL_INFO* structure into the call to define the kind of socket you want to create; however, you can create socket types that don't have an entry in the transport provider catalog. The best example of this is raw sockets under IP. Raw sockets are a form of communication that allows you to encapsulate other protocols within the UDP packet, such as the Internet Control Message Protocol (ICMP). ICMP's purpose is to deliver control, error, and informational messages among Internet hosts. Because ICMP does not provide any data transfer facilities, it is not considered to be at the same level as UDP or TCP, but at the same level as IP itself. Chapter 13 covers raw sockets in further detail.

Platform-Specific Information

Windows 95 out of the box supports the Winsock 1.1 specification. Microsoft has made freely available a Winsock 2 update that can be downloaded from its Web site (*http://www.microsoft.com/windows95/downloads/*). Also, a Winsock 2 SDK is available that includes the necessary headers and libraries to compile a Winsock 2 application. Windows 98, Windows NT 4, and Windows 2000 all support Winsock 2 natively without any necessary add-ons. Windows CE supports only the Winsock 1.1 specification.

As far as support for various transport providers goes, a few limitations should be mentioned. Windows CE supports only TCP/IP and Infrared Sockets. For Windows 95 and Windows 98, the NetBIOS transport providers (transports with the address family *AF_NETBIOS*) are not exposed to the Winsock API. If you perform a *WSAEnumProtocols*, none of the NetBIOS providers will be listed, even though they are installed on the machine. However, NetBIOS is still available by using the native NetBIOS interface, as described in Chapter 1. Last, the RSVP (which offers QOS) and ATM providers listed are natively available on Windows 98 and Windows 2000.

The Winsock API and the OSI Model

Let's look at how some of the concepts presented in this chapter relate to the OSI model (see Figure 1-1). The transport providers in the Winsock catalog that are enumerated by *WSAEnumProtocols* are at the Transport layer of the OSI model. That is, each of these transports provides a method of transferring data; however, each is a member of a protocol, and a network protocol is at the Network layer because it is the protocol that provides a method of addressing each node on a network. For example, UDP and TCP are transports, but both belong to the Internet Protocol.

The Winsock API fits between the Session and Transport layers. Winsock provides the ability to open, manipulate, and close data sessions for a given transport. Under Windows, the top three layers (Application, Presentation, and Session) relate for the most part to your Winsock application. In other words, your Winsock application controls all aspects of the session and if necessary formats the data for the purpose of your program.

Selecting the Right Protocol

When you develop a network application, you can choose from a number of available protocols to base your application on. If your application needs to communicate over a certain protocol, you don't have many choices; however, if you are developing an application from scratch, TCP/IP is the way to go, at least from the point of supportability and commitment from Microsoft. AppleTalk, NetBIOS, and IPX/SPX are protocols included by Microsoft to provide compatibility with other operating systems and were at one time the staples for network programming. This was evident when Windows 95 was installed on network computers: the default protocols installed were NetBEUI and IPX/SPX.

With the explosive growth in the Internet, the great majority of companies, educational institutions, and others have made TCP/IP their protocol of choice. In Windows 2000, Microsoft also stresses TCP/IP. Now TCP/IP is the default protocol installed, and most networking services will be based on this protocol, lessening the reliance on NetBIOS. Additionally, when you find a bug in Microsoft's implementation of TCP/IP there is a quick response to post hot fixes, whereas for bugs in other protocols there might or might not be a fix, depending on demand.

This said, TCP/IP is the safe way to go when choosing a protocol to use in a networked application. Additionally, Microsoft is showing strong support for ATM networks. If you have the luxury of developing an application that will run exclusively on ATM networks, it should be relatively safe to develop it using native ATM from Winsock. Of course, users of TCP/IP should note that ATM networks can be configured to run in TCP/IP emulation mode, which runs exceptionally well. These factors are just a few among the many to consider, in addition to the protocol characteristics needed by any application you develop.

Conclusion

In this chapter, we covered the basic characteristics to be aware of when choosing a network transport for an application. Knowledge of these characteristics is vital when it comes to successfully developing a network application based on a particular protocol. We also looked into programmatically obtaining a list of transport providers installed on a system and how to query for a particular property. Finally, we learned how to create a socket for a given network transport by specifying the correct parameters to either the *WSASocket* function or the *socket* function, and also by querying for the catalog entry using *WSAEnumProtocols* and passing in a *WSAPROTOCOL_INFO* structure to *WSASocket*. In the next chapter, we will investigate the addressing methods for each of the major protocols.

Chapter 6

Address Families and Name Resolution

In order to establish communication through Winsock, you must understand how to address a workstation using a particular protocol. This chapter explains the protocols supported by Winsock and how each protocol resolves an address specific to that family to an actual machine on the network. Winsock 2 introduces several new protocol-independent functions that can be used with sockets of any address families; in most cases, however, each address family provides its own mechanisms for resolving addresses, either through a function or as an option passed to *getsockopt*. This chapter covers only the basic knowledge necessary to form an address structure for each protocol family. Chapter 10 covers the registration and name resolution functions, which advertise a service of a given protocol family. (This is a bit different from just resolving a name.) See Chapter 10 for more details on the differences between straight name resolution and service advertising and resolution.

For each covered address family, we will discuss the basics of how to address a machine on a network. We will then create a socket for each family. Additionally, we will cover the protocol-specific options for name resolution.

IP

The Internet Protocol (IP) is commonly known as the network protocol that is used on the Internet. IP is widely available on most computer operating systems and can be used on most local area networks (LANs), such as a small network in your office, and on wide area networks (WANs), such as the Internet. By design, IP is a connectionless protocol and doesn't guarantee delivery of data. Two higher-level protocols—TCP and UDP—are used for data communication over IP.

TCP

Connection-oriented communication is accomplished through the Transmission Control Protocol (TCP). TCP provides reliable error-free data transmission between two computers. When applications communicate using TCP, a virtual connection is established between the source computer and the destination computer. Once a connection is established, data can be exchanged between the computers as a two-way stream of bytes.

UDP

Connectionless communication is accomplished through the User Datagram Protocol (UDP). UDP doesn't guarantee reliable data transmission and is capable of sending data to multiple destinations and receiving data from multiple sources. For example, if a client sends data to a server, the data is transmitted immediately, whether or not the server is ready to receive the data. If the server receives data from the client, it doesn't acknowledge the receipt. Data is transmitted using datagrams.

Both TCP and UDP use IP for data transmission and are normally referred to as TCP/IP and UDP/IP. Winsock addresses IP communication through the *AF_INET* address family, which is defined in Winsock.h and Winsock2.h.

Addressing

In IP, computers are assigned an IP address that is represented as a 32-bit quantity, formally known as an IP version 4 (IPv4) address. When a client wants to communicate with a server through TCP or UDP, it must specify the server's IP address along with a service port number. Also, when servers want to listen for incoming client requests, they must specify an IP address and a port number. In Winsock, applications specify IP addresses and service port information through the *SOCKADDR_IN* structure, which is defined as

```
struct sockaddr_in
{
    short           sin_family;
    u_short         sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8];
};
```

The *sin_family* field must be set to *AF_INET*, which tells Winsock we are using the IP address family.

> IP Version 6
>
> IP version 6 is an updated specification of IP that allows for a larger address space, which will become necessary in the near future, as IP version 4 addresses become scarce. Many of the Winsock header files contain conditional definitions for IPv6 structures; however, no current Win32

platform provides an IPv6 network stack (including Windows 2000). Microsoft Research has made available an experimental IPv6 stack that you can download and use (http://research.microsoft.com/msripv6/); however, it isn't supported, and we do not address any version 6-specific issues in this text.

The *sin_port* field defines which TCP or UDP communication port will be used to identify a server service. Applications should be particularly careful in choosing a port because some of the available port numbers are reserved for well-known services such as File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP). The ports used by well-known services are controlled and assigned by the Internet Assigned Numbers Authority (IANA) and are described in RFC 1700. Essentially, the port numbers are divided into the three ranges explained below: well-known, registered, and dynamic and/or private ports.

- 0-1023 are controlled by the IANA and are reserved for well-known services.
- 1024-49151 are registered ports listed by the IANA and can be used by ordinary user processes or programs executed by ordinary users.
- 49152-65535 are dynamic and/or private ports.

Ordinary user applications should choose the registered ports in the range 1024-49151 to avoid the possibility of using a port already in use by another application or a system service. Ports in the range 49152-65535 can also be used freely because no services are registered on these ports with the IANA. If, when using the *bind* API function, your application binds to a port that is already in use by another application on your host, the system will return the Winsock error *WSAEADDRINUSE*. Chapter 7 describes the Winsock bind process in greater detail.

The *sin_addr* field of the *SOCKADDR_IN* structure is used for storing an IP address as a 4-byte quantity, which is an unsigned long integer data type. Depending on how this field is used, it can represent a local or remote IP address. IP addresses are normally specified in Internet standard dotted notation as "a.b.c.d." Each letter represents a number for each byte and is assigned, from left to right, to the four bytes of the unsigned long integer. The final field, *sin_zero*, functions only as padding to make the *SOCKADDR_IN* structure the same size as the *SOCKADDR* structure.

A useful support function named *inet_addr* converts a dotted IP address to a 32-bit unsigned long integer quantity. The *inet_addr* function is defined as

```
unsigned long inet_addr(
    const char FAR *cp
);
```

The *cp* field is a null-terminated character string that accepts an IP address in dotted notation. Note that this function returns an IP address as a 32-bit unsigned long integer in network-byte order. (Network-byte order is described shortly, under "Byte ordering.")

Special addresses

Two special IP addresses affect the behavior of a socket in certain situations. The special address *INADDR_ANY* allows a server application to listen for client activity over every network interface on a host computer. Typically, server applications use this address when they bind a socket to the local interface to listen for connections. If you have a multihomed system, this address allows a single application to accept responses from multiple interfaces.

The special address *INADDR_BROADCAST* can be used to send broadcast UDP datagrams over an IP network. Using this special address requires an application to set the socket option *SO_BROADCAST*. Chapter 9 explains this option in greater detail.

Byte ordering

Different computer processors represent numbers in *big-endian* and *little-endian* form, depending on how they are designed. For example, on Intel x86 processors, multibyte numbers are represented in little-endian form: the bytes are ordered from least significant byte to most significant byte. When an IP address and port number are specified as multibyte quantities in a computer, they are represented in *host-byte* order. However, when IP addresses and port numbers are specified over a network, Internet networking standards specify that multibyte values must be represented in big-endian form (most significant byte to least significant byte), normally referred to as *network-byte* order.

A series of functions can be used to convert a multibyte number from host-byte order to network-byte order and vice versa. The following four API functions convert a number from host-byte to network-byte order:

```
u_long htonl(u_long hostlong);

int WSAHtonl(
    SOCKET s,
    u_long hostlong,
    u_long FAR * lpnetlong
);

u_short htons(u_short hostshort);

int WSAHtons(
    SOCKET s,
    u_short hostshort,
    u_short FAR * lpnetshort
);
```

The *hostlong* parameter of *htonl* and *WSAHtonl* is a 4-byte number in host-byte order. The *htonl* function returns the number in network-byte order, whereas the *WSAHtonl* function returns the number in network-byte order through the *lpnetlong* parameter. The *hostshort* parameter of *htons* and *WSAHtons* is a 2-byte number in host-byte order. The *htons* function returns the number as a 2-byte value in network-byte order, whereas the *WSAHtons* function returns the number through the *lpnetshort* parameter.

The next four functions are the opposite of the preceding four functions: they convert from network-byte order to host-byte order.

```
u_long ntohl(u_long netlong);

int WSANtohl(
    SOCKET s,
    u_long netlong,
    u_long FAR * lphostlong
);

u_short ntohs(u_short netshort);

int WSANtohs(
    SOCKET s,
    u_short netshort,
    u_short FAR * lphostshort
);
```

We will now demonstrate how to create a *SOCKADDR_IN* structure using the *inet_addr* and *htons* functions described above.

```
SOCKADDR_IN InternetAddr;
INT nPortId = 5150;

InternetAddr.sin_family = AF_INET;

// Convert the proposed dotted Internet address 136.149.3.29
// to a 4-byte integer, and assign it to sin_addr

InternetAddr.sin_addr.s_addr = inet_addr("136.149.3.29");

// The nPortId variable is stored in host-byte order. Convert
// nPortId to network-byte order, and assign it to sin_port.

InternetAddr.sin_port = htons(nPortId);
```

Now that you have the basics of addressing IP through a *SOCKADDR_IN* structure, you can prepare to set up communication for TCP or UDP by creating a socket.

Creating a Socket

Creating an IP socket offers applications the capability to communicate over the TCP, UDP, and IP protocols. To open an IP socket using the TCP protocol, call the *socket* function or the *WSASocket* function with the address family *AF_INET* and the socket type *SOCK_STREAM*, and set the protocol field to 0, as follows:

```
s = socket(AF_INET, SOCK_STREAM, 0);

s = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED);
```

To open an IP socket using the UDP protocol, simply specify the socket type *SOCK_DGRAM* instead of *SOCK_STREAM* in the *socket* and *WSASocket* calls above. It is also possible to open a socket to communicate directly over IP. This is accomplished by setting the socket type to *SOCK_RAW*. Chapter 13 describes the *SOCK_RAW* option in greater detail.

Name Resolution

When a Winsock application wants to communicate with a host over IP, it must know the host's IP address. From an application user's point of view, IP addresses aren't easy to remember. Most people would much rather refer to a machine by using an easy-to-remember, user-friendly host name instead of an IP address. Winsock provides two support functions that can help you resolve a host name to an IP address.

The Windows Sockets *gethostbyname* and *WSAAsyncGetHostByName* API functions retrieve host information corresponding to a host name from a host database. Both functions return a *HOSTENT* structure that is defined as

```
struct hostent
{
    char FAR *       h_name;
    char FAR * FAR * h_aliases;
    short            h_addrtype;
    short            h_length;
    char FAR * FAR * h_addr_list;
};
```

The *h_name* field is the official name of the host. If your network uses the Domain Name System (DNS), it is the Fully Qualified Domain Name (FQDN) that causes the name server to return a reply. If your network uses a local "hosts" file, it is the first entry after the IP address. The *h_aliases* field is a null-terminated array of alternative names for the host. The *h_addrtype* represents the address family being returned. The *h_length* field defines the length in bytes of each address in the *h_addr_list* field. The *h_addr_list* field is a null-terminated array of IP addresses for the host. (A host can have more than one IP address assigned to it.) Each address in the array is returned in network-byte order. Normally, applications use the first address in the array. However, if more than one address is returned, applications should randomly choose an available address rather than always use the first address.

The *gethostbyname* API function is defined as

```
struct hostent FAR * gethostbyname (
    const char FAR *  name
);
```

The *name* parameter represents a friendly name of the host you are looking for. If this function succeeds, a pointer to a *HOSTENT* structure is returned. Note that the memory where the *HOSTENT* structure is stored is system memory. An application shouldn't rely on this to remain static. Since this memory is maintained by the system, your application doesn't have to free the returned structure.

The *WSAAsyncGetHostByName* API function is an asynchronous version of the *gethostbyname* function that uses Windows messages to inform an application when this function completes. *WSAAsyncGetHostByName* is defined as

```
HANDLE WSAAsyncGetHostByName(
    HWND hWnd,
    unsigned int wMsg,
    const char FAR * name,
    char FAR * buf,
    int buflen
);
```

The *hWnd* parameter is the handle of the window that will receive a message when the asynchronous request completes. The *wMsg* parameter is the Windows message to be received when the asynchronous request completes. The *name* parameter represents a user-friendly name of the host we are looking for. The *buf* parameter is a pointer to the data area to receive the *HOSTENT* data. This buffer must be larger than a *HOSTENT* structure and should be set to the size defined in *MAXGETHOSTSTRUCT*.

Two more functions that retrieve host information are worth mentioning: the *gethostbyaddr* and *WSAAsyncGetHostByAddr* API functions, which are designed to retrieve host information corresponding to an IP network address. These functions are useful when you have the IP address of a host and want to look up its user-friendly name. The *gethostbyaddr* function is defined as

```
struct HOSTENT FAR * gethostbyaddr(
    const char FAR * addr,
    int len,
    int type
);
```

The *addr* parameter is a pointer to an IP address in network-byte order. The *len* parameter specifies the byte length of the *addr* parameter. The *type* parameter should specify the value *AF_INET*, which indicates that this is an IP address type. The *WSAAsyncGetHostByAddr* API function is an asynchronous version of *gethostbyaddr*.

Port numbers

In addition to the IP address of a remote computer, an application must know the service's port number to communicate with a service running on a local or remote computer. When using TCP and UDP, applications must decide which ports they plan to communicate over. There are well-known port numbers reserved by server services that support protocols of a level higher than TCP. For example, port 21 is reserved for FTP, and port 80 is reserved for HTTP. As mentioned earlier, well-known services typically use ports 1-1023 to set up communication. If you are developing a TCP application that doesn't use one of the well-known services, consider using ports above 1023 to avoid using a port already being used. You can retrieve port numbers for well-known services by calling the *getservbyname* and *WSAAsyncGetServByName* functions. These functions simply retrieve static information from a file named *services*. In Windows 95 and Windows 98, the services file is located under %WINDOWS%; and in Windows NT and Windows 2000, it is located under %WINDOWS%\System32\Drivers\Etc. The *getservbyname* function is defined as follows:

```
struct servent FAR * getservbyname(
    const char FAR * name,
    const char FAR * proto
);
```

The *name* parameter represents the name of the service you are looking for. For example, if you are trying to locate the port for FTP, you should set the *name* parameter to point to the string "ftp." The *proto* parameter optionally points to a string that indicates the protocol that the service in *name* is registered under. The *WSAAsyncGetServByName* function is an asynchronous version of *getservbyname*.

Windows 2000 has a new dynamic method to register and query service information for TCP and UDP. Server applications can register the service name, IP address, and port number of a service by using the *WSASetService* function. Client applications can query this service information by using a combination of the following API functions: *WSALookupServiceBegin*, *WSALookupServiceNext*, and *WSALookupServiceEnd*. Chapter 10 covers the details of using these capabilities.

Infrared Sockets

Infrared sockets, or IrSock, are an exciting new technology first introduced on the Windows CE platform. Infrared sockets allow two PCs to communicate with each other through an infrared serial port. Infrared sockets are now available on Windows 98 and Windows 2000. Infrared sockets differ from traditional sockets in that infrared sockets are designed to take into account the transient nature of portable computing. Infrared sockets present a new name resolution model that will be discussed in the next section.

Addressing

Because most computers with Infrared Data Association (IrDA) devices are likely to move around, traditional name-resolution schemes don't work well. Conventional resolution methods assume static resources such as name servers, which cannot be used when a person is moving a handheld PC or laptop computer running a network client. To circumvent this problem, IrSock is designed to browse in-range resources in an ad hoc manner without the overhead of a large network, and it doesn't use standard Winsock name service functions or even IP addressing. Instead, the name service has been incorporated into the communication stream, and a new address family has been introduced to support services bound to infrared serial ports. The IrSock address structure includes a service name that describes the application used in bind and connect calls, and a device identifier that describes the device on which the service runs. This pair is analogous to the IP address and port number tuple used by conventional TCP/IP sockets. The IrSock address structure is defined as

```
typedef struct sockaddr_irda {
    u_short    irdaAddressFamily;
    u_char     irdaDeviceID[4];
    char       irdaServiceName[25];
} SOCKADDR_IRDA;
```

The *irdaAddressFamily* field is always set to *AF_IRDA*. The *irdaDeviceID* is a four-character string that uniquely identifies the device on which a particular service is running. This field is ignored when an IrSock server is created. However, the field is significant for a client because it specifies the IrDA device to connect to. (There can be multiple devices in range.) Finally, the *irdaServiceName* field is the name of the service that the application either will register itself with or is trying to connect to.

Name Resolution

Addressing can be based on IrDA Logical Service Access Point Selectors (LSAP-SELs) or on services registered with the Information Access Services (IAS). The IAS abstracts a service from an LSAP-SEL into a user-friendly text service name, in much the same way that an Internet domain name server maps names to numeric IP addresses. You can use either an LSAP-SEL or a user-friendly name to successfully connect, but user-friendly names require name resolution. For the most part, you shouldn't use the direct LSAP-SEL "address" because the address space for IrDA services is limited. The Win32 implementation allows LSAP-SEL integer identifiers in the range of 1 to 127. Essentially an IAS server can be thought of as a WINS server because it associates an LSAP-SEL with a textual service name.

An actual IAS entry has three fields of importance: class name, attribute, and attribute value. For example, let's say a server wishes to register itself under the service name *MyServer*. This is accomplished when the server issues the bind call with the appropriate *SOCKADDR_IRDA* structure. Once this occurs, an IAS entry is added with the class name *MyServer*, the attribute IrDA:TinyTP:LsapSel, and an attribute value of, say, 3. The attribute value is the next unused LSAP-SEL assigned by the system upon registration. The client, on the other hand, passes in a *SOCKADDR_IRDA* structure to the connect call. This initiates an IAS lookup for a service with the class name *MyServer* and the attribute IrDA:TinyTP:LsapSel. The IAS query will return the value 3. You can formulate your own IAS query by using the socket option *IRLMP_IAS_QUERY* in the *getsockopt* call.

If you want to bypass IAS altogether (which is not recommended), you can specify an LSAP-SEL address directly for a server name or an endpoint to which a client wants to connect. You should bypass IAS only to communicate with legacy IrDA devices that don't provide any kind of IAS registration (such as infrared-capable printers). You

can bypass the IAS registration and lookup by specifying the service name in the *SOCKADDR_IRDA* structure as LSAP-SEL-*xxx*, where *xxx* is the attribute value between 1 and 127. For a server, this would directly assign the server to the given LSAP-SEL address (assuming the LSAP-SEL address is unused). For a client, this bypasses the IAS lookup and causes an immediate attempt to connect to whatever service is running on that LSAP-SEL.

Enumerating IrDA Devices

Because infrared devices move in and out of range, a method of dynamically listing all available infrared devices within range is necessary. This section describes how to accomplish that. Let's begin with a few platform discrepancies between the Windows CE implementation and the Windows 98 and Windows 2000 implementation. Windows CE supported IrSock before the other platforms and provided minimal information about infrared devices. Later, Windows 98 and Windows 2000 provided support for IrSock, but they added additional "hint" information returned by the enumeration request. (This hint information will be discussed shortly.) As a result, the Af_irda.h header file for Windows CE contains the original, minimal structure definitions; however, the new header file for the other platforms contains conditional structure definitions for each platform that now supports IrSock. We recommend that you use the later Af_irda.h header file for consistency.

The way to enumerate nearby infrared devices is by the *IRLMP_ENUM_DEVICES* command for *getsockopt*. A *DEVICELIST* structure is passed as the *optval* parameter. There are two structures, one for Windows 98 and Windows 2000 and one for Windows CE. They are defined as

```
typedef struct _WINDOWS_DEVICELIST
{
    ULONG                       numDevice;
    WINDOWS_IRDA_DEVICE_INFO    Device[1];
} WINDOWS_DEVICELIST, *PWINDOWS_DEVICELIST, FAR *LPWINDOWS_DEVICELIST;

typedef struct _WCE_DEVICELIST
{
    ULONG                       numDevice;
    WCE_IRDA_DEVICE_INFO        Device[1];
} WCE_DEVICELIST, *PWCE_DEVICELIST;
```

The only difference between the Windows 98 and Window 2000 structure and the Windows CE structure is that the Windows 98 and Windows 2000 structure contains an array of *WINDOWS_IRDA_DEVICE_INFO* structures as opposed to an array of *WCE_IRDA_DEVICE_INFO* structures. A conditional *#define* directive declares *DEVICELIST* as the appropriate structure depending on the target platform. Likewise, two declarations for the *IRDA_DEVICE_INFO* structures exist:

```
typedef struct _WINDOWS_IRDA_DEVICE_INFO
{
    u_char  irdaDeviceID[4];
    char    irdaDeviceName[22];
    u_char  irdaDeviceHints1;
    u_char  irdaDeviceHints2;
    u_char  irdaCharSet;
} WINDOWS_IRDA_DEVICE_INFO, *PWINDOWS_IRDA_DEVICE_INFO,
  FAR *LPWINDOWS_IRDA_DEVICE_INFO;

typedef struct _WCE_IRDA_DEVICE_INFO
{
    u_char  irdaDeviceID[4];
    char    irdaDeviceName[22];
    u_char  Reserved[2];
} WCE_IRDA_DEVICE_INFO, *PWCE_IRDA_DEVICE_INFO;
```

Again, a conditional *#define* directive declares *IRDA_DEVICE_INFO* to the correct structure definition depending on the target platform.

As mentioned earlier, the function to use for the actual enumeration of infrared devices is *getsockopt* with the option *IRLMP_ENUM_DEVICES*. The following piece of code lists the device IDs of all infrared devices nearby.

```
SOCKET      sock;
DEVICELIST  devList;
DWORD       dwListLen=sizeof(DEVICELIST);

sock = WSASocket(AF_IRDA, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED);
...
devList.numDevice = 0;
dwRet = getsockopt(sock, SOL_IRLMP, IRLMP_ENUMDEVICES,
    (char *)&devList, &dwListLen);
```

Before you pass a *DEVICELIST* structure into the *getsockopt* call, don't forget to set the *numDevice* field to 0. A successful enumeration will set the *numDevice* field to a value greater than 0 and set an equal number of *IRDA_DEVICE_INFO* structures in the *Device* field. Also, in an actual application you probably want to perform *getsockopt* more than once in order to check for devices that just moved into range. For example, attempting to discover an infrared device in five tries or less is a good heuristic. Simply place the call in a loop with a short call to the *Sleep* function after each unsuccessful enumeration.

Now that you know how to enumerate infrared devices, creating a client or a server is simple. The server side of the equation is a bit simpler because it looks like a "normal" server. That is, no extra steps are required. The general steps for an IrSock server are

1. Create a socket of address family *AF_IRDA* and socket type *SOCK_STREAM*.
2. Fill out a *SOCKADDR_IRDA* structure with the service name of the server.
3. Call *bind* with the socket handle and the *SOCKADDR_IRDA* structure.
4. Call *listen* with the socket handle and the backlog limit.

5. Block on an *accept* call for incoming clients.

The steps for a client are a bit more involved, as you must enumerate infrared devices. The following steps are necessary for an IrSock client:

1. Create a socket of address family *AF_IRDA* and socket type *SOCK_STREAM*.
2. Enumerate available infrared devices by calling *getsockopt* with the *IRLMP_ENUM_DEVICES* option.
3. For each device returned, fill out a *SOCKADDR_IRDA* structure with the device ID returned and the service name you want to connect to.
4. Call the *connect* function with the socket handle and the *SOCKADDR_IRDA* structure. Do this for each structure filled out in step 3 until a connect succeeds.

Querying IAS

There are two ways to find out whether a given service is running on a particular device. The first method is to actually attempt a connection to the service; the other is to query IAS for the given service name. Both methods require you to enumerate all infrared devices, and attempt a query (or connection) with each device until one of them succeeds or you have exhausted every device. Perform a query by calling *getsockopt* with the *IRLMP_IAS_QUERY* option. A pointer to an *IAS_QUERY* structure is passed as the *optval* parameter. Again, there are two *IAS_QUERY* structures, one for Windows 98 and Windows 2000, and another for Windows CE. Here are the definitions of each structure:

```
typedef struct _WINDOWS_IAS_QUERY
{
     u_char    irdaDeviceID[4];
     char      irdaClassName[IAS_MAX_CLASSNAME];
     char      irdaAttribName[IAS_MAX_ATTRIBNAME];
     u_long    irdaAttribType;
     union
    {
        LONG   irdaAttribInt;
        struct
        {
            u_long   Len;
            u_char   OctetSeq[IAS_MAX_OCTET_STRING];
        } irdaAttribOctetSeq;
        struct
        {
            u_long   Len;
            u_long   CharSet;
            u_char   UsrStr[IAS_MAX_USER_STRING];
        } irdaAttribUsrStr;
    } irdaAttribute;
} WINDOWS_IAS_QUERY, *PWINDOWS_IAS_QUERY,
  FAR *LPWINDOWS_IAS_QUERY;

typedef struct _WCE_IAS_QUERY
{
    u_char     irdaDeviceID[4];
    char       irdaClassName[61];
```

```
    char       irdaAttribName[61];
    u_short    irdaAttribType;
    union
    {
        int    irdaAttribInt;
        struct
        {
            int      Len;
            u_char   OctetSeq[1];
            u_char   Reserved[3];
        } irdaAttribOctetSeq;
        struct
        {
            int      Len;
            u_char   CharSet;
            u_char   UsrStr[1];
            u_char   Reserved[2];
        } irdaAttribUsrStr;
    } irdaAttribute;
} WCE_IAS_QUERY, *PWCE_IAS_QUERY;
```

As you can see, the two structure definitions are similar except for the lengths of certain character arrays.

Performing a query for the LSAP-SEL number of a particular service is simple: set the *irdaClassName* field to the property string for *LSAP-SELs*, which is "IrDA:IrLMP:LsapSel", and set the *irdaAttributeName* field to the service name you want to query for. Additionally, you have to set the *irdaDeviceID* field with a valid device within range.

Creating a Socket

Creating an infrared socket is simple. Few options are required because IrSock supports only connection-oriented streams. The following code illustrates how to create an infrared socket using either the *socket* or the *WSASocket* call. You must use *socket* for Windows CE because of its Winsock 1.1 limitation.

```
s = socket(AF_IRDA, SOCK_STREAM, 0);

s = WSASocket(AF_IRDA, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED);
```

If you want to be specific, you can pass *IRDA_PROTO_SOCK_STREAM* as the protocol parameter of either function. However, the protocol parameter isn't required because the transport catalog has only one entry of address family *AF_IRDA*. Specifying *AF_IRDA* causes that transport entry to be used by default.

Socket Options

Many *SO_* socket options aren't meaningful to IrDA. Only *SO_LINGER* is specifically supported. The IrSock-specific socket options are of course supported only on sockets of the address family *AF_IRDA*. These options are also covered in Chapter 9, which summarizes all socket options and their parameters.

IPX/SPX

The Internetwork Packet Exchange (IPX) protocol is commonly known as the protocol used with computers featuring Novell NetWare client/server networking services. IPX provides connectionless communication between two processes; therefore, if a workstation transmits a data packet, there is no guarantee that the packet will be delivered to the destination. If an application needs guaranteed delivery of data and insists on using IPX, it can use a higher-level protocol over IPX, such as the Sequence Packet Exchange (SPX) and SPX II protocols, in which SPX packets are transmitted through IPX. Winsock provides applications with the capability to communicate through IPX on Windows 95, Windows 98, Windows NT, and Windows 2000 but not on Windows CE.

Addressing

In an IPX network, network segments are bridged together using an IPX router. Each network segment is assigned a unique 4-byte network number. As more network segments are bridged together, IPX routers manage communication between different network segments using the unique network segment numbers. When a computer is attached to a network segment, it is identified using a unique 6-byte node number, which is usually the physical address of the network adapter. A node (which is a computer) is typically capable of having one or more processes forming communication over IPX. IPX uses socket numbers to distinguish communication for processes on a node.

To prepare a Winsock client or server application for IPX communication, you have to set up a *SOCKADDR_IPX* structure. The *SOCKADDR_IPX* structure is defined in the Wsipx.h header file, and your application must include this file after including Winsock2.h. The *SOCKADDR_IPX* structure is defined as

```
typedef struct sockaddr_ipx
{
    short          sa_family;
    char           sa_netnum[4];
    char           sa_nodenum[6];
    unsigned short sa_socket;
} SOCKADDR_IPX, *PSOCKADDR_IPX, FAR *LPSOCKADDR_IPX;
```

The *sa_family* field should always be set to the *AF_IPX* value. The *sa_netnum* field is a 4-byte number representing a network number of a network segment on an IPX network. The *sa_nodenum* field is a 6-byte number representing a node number of a computer's physical address. The *sa_socket* field represents a socket or port used to distinguish IPX communication on a single node.

Creating a Socket

Creating a socket using IPX offers several possibilities. To open an IPX socket, call the *socket* function or the *WSASocket* function with the address family *AF_IPX*, the socket type *SOCK_DGRAM*, and the protocol *NSPROTO_IPX*, as follows:

```
s = socket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX);

s = WSASocket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX,
    NULL, 0, WSA_FLAG_OVERLAPPED);
```

Note that the third parameter protocol must be specified and cannot be 0. This is important because this field can be used to set specific IPX packet types.

As we mentioned earlier, IPX provides unreliable connectionless communication using datagrams. If an application needs reliable communication using IPX, it can use higher-level protocols over IPX, such as SPX and SPX II. This can be accomplished by setting the type and protocol fields of the *socket* and *WSASocket* calls to the socket type *SOCK_SEQPACKET* or *SOCK_STREAM*, and the protocol *NSPROTO_SPX* or *NSPROTO_SPXII*.

If *SOCK_STREAM* is specified, data is transmitted as a continuous stream of bytes with no message boundaries—similar to the behavior of sockets in TCP/IP. On the other hand, if *SOCK_SEQPACKET* is specified, data is transmitted with message boundaries. For example, if a sender transmits 2000 bytes, the receiver won't return until all 2000 bytes have arrived. SPX and SPX II accomplish this by setting an end-of-message bit in an SPX header. When *SOCK_SEQPACKET* is specified, this bit is respected—meaning Winsock *recv* and *WSARecv* calls won't complete until a packet is received with this bit set. If *SOCK_STREAM* is specified, the end-of-message bit isn't respected, and *recv* completes as soon as any data is received, regardless of the setting of the end-of-message bit. From the sender's perspective (using the *SOCK_SEQPACKET* type), sends smaller than a single packet are always sent with the endof-message bit set. Sends larger than single packets are packetized with the end-ofmessage bit set on only the last packet of the send.

Binding a socket

When an IPX application associates a local address with a socket using *bind*, you shouldn't specify a network number and a node address in a *SOCKADDR_IPX* structure. The *bind* function populates these fields using the first IPX network interface available on the system. In the case of a machine with multiple network interfaces (a multihomed machine), it isn't necessary to bind to a specific interface. Windows 95, Windows 98, Windows NT, and Windows 2000 provide a virtual internal network in which each network interface can be reached regardless of the physical network it is attached to. We will describe internal network numbers in greater detail later in this chapter. After your application successfully binds to a local interface, you can retrieve local network number and node number information using the *getsockname* function, as in the following code fragment:

```
SOCKET sdServer;
SOCKADDR_IPX IPXAddr;
int addrlen = sizeof(SOCKADDR_IPX);

if ((sdServer = socket (AF_IPX, SOCK_DGRAM, NSPROTO_IPX))
    == INVALID_SOCKET)
{
    printf("socket failed with error %d\n",
        WSAGetLastError());
    return;
}

ZeroMemory(&IPXAddr, sizeof(SOCKADDR_IPX));
IPXAddr.sa_family = AF_IPX;
IPXAddr.sa_socket = htons(5150);

if (bind(sdServer, (PSOCKADDR) &IPXAddr, sizeof(SOCKADDR_IPX))
    == SOCKET_ERROR)
{
    printf("bind failed with error %d\n",
        WSAGetLastError());
    return;
}

if (getsockname((unsigned) sdServer, (PSOCKADDR) &IPXAddr, &addrlen)
    == SOCKET_ERROR)
{
    printf("getsockname failed with error %d",
```

```
        WSAGetLastError());
    return;
}

// Print out SOCKADDR_IPX information returned from
// getsockname()
```

Network number vs. internal network number

A network number (known as an external network number) identifies network segments in IPX and is used for routing IPX packets between network segments. Windows 95, Windows 98, Windows NT, and Windows 2000 also feature an internal network number that is used for internal routing purposes and to uniquely identify the computer on an inter-network (several networks bridged together). The internal network number is also known as a virtual network number—the internal network number identifies another (virtual) segment on the inter-network. Thus, if you configure an internal network number for a computer running Windows 95, Windows 98, Windows NT, or Windows 2000, a NetWare server or an IPX router will add an extra hop in its route to that computer.

The internal virtual network serves a special purpose in the case of a multihomed computer. When applications bind to a local network interface, they shouldn't specify local interface information but instead should set the *sa_netnum* and *sa_nodenum* fields of a *SOCKADDR_IPX* structure to 0. This is because IPX is able to route packets from any external network to any of the local network interfaces using the internal virtual network. For example, even if your application explicitly binds to the network interface on Network A, and a packet comes in on Network B, the internal network number will cause the packet to be routed internally so that your application receives it.

Setting IPX packet types through Winsock

Winsock allows your application to specify IPX packet types when you create a socket using the *NSPROTO_IPX* protocol specification. The packet type field in an IPX packet indicates the type of service offered or requested by the IPX packet. In Novell, the following IPX packet types are defined:

- 01h Routing Information Protocol (RIP) Packet
- 04h Service Advertising Protocol (SAP) Packet
- 05h Sequenced Packet Exchange (SPX) Packet
- 11h NetWare Core Protocol (NCP) Packet
- 14h Propagated Packet for Novell NetBIOS

To modify the IPX packet type, simply specify *NSPROTO_IPX* + *n* as the protocol parameter of the *socket* API, with *n* representing the packet type number. For example, to open an IPX socket that sets the packet type to 04h (SAP Packet), use the following *socket* call:

```
s = socket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX + 0x04);
```

Name Resolution

As you can probably tell, addressing IPX in Winsock is sort of ugly since you must supply multibyte network and node numbers to form an address. IPX provides applications with the ability to locate services by using user-friendly names to retrieve network number, node number, and port number in an IPX network through the SAP protocol. As we will see in Chapter 10, Winsock 2 provides a protocol-independent method for name registration using the *WSASetService* API function. Through the SAP protocol, IPX server applications can use *WSASetService* to register under a user-friendly name the network number, node number, and port number they are listening on.

Winsock 2 also provides a protocol-independent method of name resolution through the following API functions: *WSALookupServiceBegin*, *WSALookupServiceNext*, and *WSALookupServiceEnd*.

It is possible to perform your own name-service registration and lookups by opening an IPX socket and specifying an SAP packet type. After opening the socket, you can begin broadcasting SAP packets to the IPX network to register and locate services on the network. This requires that you understand the SAP protocol in great detail and that you deal with the programming details of decoding an IPX SAP packet.

NetBIOS

The NetBIOS address family is yet another protocol family accessible from Winsock. You will be familiar with many of the topics and caveats discussed here from the NetBIOS discussion in Chapter 1. Addressing NetBIOS from Winsock still requires the knowledge of NetBIOS names and LANA numbers. We'll assume you've read those sections in Chapter 1, and we'll continue on with the specifics of accessing NetBIOS from Winsock.

> NOTE
>
> The NetBIOS address family is exposed by Winsock only on Windows NT and Windows 2000. It is not available on the Windows 95 and Windows 98 platforms or on Windows CE.

Addressing

The basis for addressing a machine under NetBIOS is a NetBIOS name, which we covered in Chapter 1. To review, a NetBIOS name is 16 characters long, with the last character reserved as a qualifier to define what type of service the name belongs to. There are two types of NetBIOS names: unique and group. A unique name can be registered by only one process on the entire network. For example, a session-based server would register the name FOO, and clients who wanted to contact that server would attempt a connection to FOO. Group names allow a group of applications to register the same name, so datagrams sent to that name will be received by all processes that registered that name.

In Winsock, the NetBIOS addressing structure is defined in Wsnetbs.h, as follows:

```
#define NETBIOS_NAME_LENGTH 16

typedef struct sockaddr_nb
{
    short   snb_family;
    u_short snb_type;
    char    snb_name[NETBIOS_NAME_LENGTH];
} SOCKADDR_NB, *PSOCKADDR_NB, FAR *LPSOCKADDR_NB;
```

The *snb_family* field specifies the address family of this structure and should always be set to *AF_NETBIOS*. The *snb_type* field is used to specify a unique or a group name. The following defines can be used for this field:

```
#define NETBIOS_UNIQUE_NAME         (0x0000)
#define NETBIOS_GROUP_NAME          (0x0001)
```

Finally, the *snb_name* field is the actual NetBIOS name.

Now that you know what each field means and what it should be set to, the following handy macro defined in the header file sets all of this for you:

```
#define SET_NETBIOS_SOCKADDR(_snb, _type, _name, _port)       \
    {                                                         \
        int _i;                                               \
        (_snb)->snb_family = AF_NETBIOS;                      \
        (_snb)->snb_type = (_type);                           \
        for (_i = 0; _i < NETBIOS_NAME_LENGTH - 1; _i++) {    \
            (_snb)->snb_name[_i] = ' ';                       \
        }                                                     \
        for (_i = 0;                                          \
             *((_name) + _i) != '\0'                          \
             && _i < NETBIOS_NAME_LENGTH - 1;                 \
             _i++)                                            \
        {                                                     \
            (_snb)->snb_name[_i] = *((_name)+_i);             \
        }                                                     \
        (_snb)->snb_name[NETBIOS_NAME_LENGTH - 1] = (_port);  \
    }
```

The first parameter to the macro, *_snb*, is the address of the *SOCKADDR_NB* structure you are filling in. As you can see, it automatically sets the *snb_family* field to *AF_NETBIOS*. For the *_type* parameter to the macro, specify *NETBIOS_UNIQUE_NAME* or *NETBIOS_GROUP_NAME*. The *_name* parameter is the NetBIOS name. The macro assumes it is either at least *NETBIOS_NAME_LENGTH* _ 1 characters in length or is null-terminated if shorter. Notice that the *snb_name* field is prefilled with spaces. Finally, the macro sets the 16th character of the *snb_name* character string to the value of the *_port* parameter.

You can see that the NetBIOS name structure in Winsock is straightforward and shouldn't present any particular difficulties. The name resolution is performed under the hood, so unlike with TCP and IrDA, you don't have to resolve a name into a physical address before any operations. This becomes clear when you consider that NetBIOS is implemented over multiple protocols and each protocol has its own addressing scheme. In the next chapter, we'll present a simple client/server using the NetBIOS interface in Winsock.

Creating a Socket

The most important consideration when you create a NetBIOS socket is the LANA number. Just as in the native NetBIOS API, you have to be aware of which LANA numbers concern your application. Remember that in order for a NetBIOS client and server to communicate, they must have a common transport protocol on which they both listen or connect. There are two ways to create a NetBIOS socket. The first is to call *socket* or *WSASocket*, as follows:

```
s = WSASocket(AF_NETBIOS, SOCK_DGRAM | SOCK_SEQPACKET, -lana,
              NULL, 0, WSA_FLAG_OVERLAPPED);
```

The *type* parameter of *WSASocket* is either *SOCK_DGRAM* or *SOCK_SEQPACKET* (don't specify both), depending on whether you want a connectionless datagram or a connection-oriented session socket. The third parameter, *protocol*, is the LANA number on which the socket should be created, except that you have to make it negative. The fourth parameter is null because you are specifying your own parameters, not using a *WSAPROTOCOL_INFO* structure. The fifth parameter isn't used. Finally, the *dwFlags* parameter is set to *WSA_FLAG_OVERLAPPED*; you should specify *WSA_FLAG_OVERLAPPED* on all calls to *WSASocket*.

The drawback to the first method of socket creation is that you need to know which LANA numbers are valid to begin with. Unfortunately, Winsock doesn't have a nice, short method of enumerating available LANA numbers. The alternative in Winsock is to enumerate all transport protocols with *WSAEnumProtocols*. Of course, you could call the *Netbios* function with the *NCBENUM* command to get the valid LANAs. Chapter 5 described how to call

WSAEnumProtocols. The following sample enumerates all transport protocols, searches for a NetBIOS transport, and creates a socket for each one.

```
dwNum = WSAEnumProtocols(NULL, lpProtocolBuf, &dwBufLen);
if (dwNum == SOCKET_ERROR)
{
    // Error
}
for (i = 0; i < dwNum; i++)
{
    // Look for those entries in the AF_NETBIOS address family
    if (lpProtocolBuf[i].iAddressFamily == AF_NETBIOS)
    {
        // Look for either SOCK_SEQPACKET or SOCK_DGRAM
        if (lpProtocolBuf[i].iSocketType == SOCK_SEQPACKET)
        {
             s[j++] = WSASocket(FROM_PROTOCOL_INFO,
                FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
                   &lpProtocolBuf[i], 0, WSA_FLAG_OVERLAPPED);
        }
    }
}
```

In the above pseudocode, we enumerate the available protocols and iterate through them looking for those belonging to the *AF_NETBIOS* address family. Next we check the socket type, and in this case, look for entries of type *SOCK_SEQPACKET*. Otherwise, if we wanted datagrams we would check for *SOCK_DGRAM*. If this matches, we have a NetBIOS transport we can use. If you need the LANA number, take the absolute value of the *iProtocol* field in the *WSAPROTOCOL_INFO* structure. The only exception is for LANA 0. The *iProtocol* field for this LANA is 0x80000000 because 0 is reserved for use by Winsock. The variable *j* will contain the number of valid transports.

AppleTalk

AppleTalk support in Winsock has been around for a while, although few people are aware of it. AppleTalk probably won't be a protocol you choose to use unless you are communicating with Macintosh computers. AppleTalk is somewhat similar to NetBIOS in that it is name-based on a per-process basis. That is, a server dynamically registers a particular name that it will be known as. Clients use this name to establish a connection. However, AppleTalk names are substantially more complicated than NetBIOS names. The next section will discuss how computers using the AppleTalk protocol are addressed on the network.

Addressing

An AppleTalk name is actually based on three separate names: the name, the type, and the zone. Each name can be up to 32 characters in length. The name identifies the process and its associated socket on a machine. The type is a subgrouping mechanism for zones. Traditionally, a zone is a network of AppleTalk-enabled computers physically located on the same loop. Microsoft's implementation of AppleTalk allows a Windows machine to specify the default zone it is located within. Multiple networks can be bridged together. These human-friendly names map to a socket number, a node number, and a network number. An AppleTalk name must be unique within the given type and zone. This requirement is enforced by the Name Binding Protocol (NBP), which broadcasts a query to see whether the name is already in use. Under the hood, AppleTalk uses the Routing Table Maintenance Protocol (RTMP) to dynamically discover routes to the different AppleTalk networks linked together.

The following structure provides the basis for addressing AppleTalk hosts from Winsock:

```
typedef struct sockaddr_at
{
    USHORT  sat_family;
    USHORT  sat_net;
    UCHAR   sat_node;
    UCHAR   sat_socket;
} SOCKADDR_AT, *PSOCKADDR_AT;
```

Notice that the address structure contains only characters or short integers and not friendly names. The *SOCKADDR_AT* structure is passed into Winsock calls such as *bind*, *connect*, and *WSAConnect*, but in order to translate the human-readable names you must query the network to either resolve or register that name first. This is done by using a call to *getsockopt* or *setsockopt*, respectively.

Registering an AppleTalk name

A server that wants to register a particular name so that clients can easily connect to it calls *setsockopt* with the *SO_REGISTER_NAME* option. For all socket options involving AppleTalk names, use the *WSH_NBP_NAME* structure, which is defined as

```
typedef struct
{
    CHAR    ObjectNameLen;
    CHAR    ObjectName[MAX_ENTITY];
    CHAR    TypeNameLen;
    CHAR    TypeName[MAX_ENTITY];
    CHAR    ZoneNameLen;
    CHAR    ZoneName[MAX_ENTITY];
} WSH_NBP_NAME, *PWSH_NBP_NAME;
```

A number of types—such as *WSH_REGISTER_NAME*, *WSH_DEREGISTER_NAME*, and *WSH_REMOVE_NAME*—are defined based on the *WSH_NBP_NAME* structure. Using the appropriate type depends on whether you look up a name, register a name, or remove a name.

The following code sample illustrates how to register an AppleTalk name.

```
#define MY_ZONE    "*"
#define MY_TYPE    "Winsock-Test-App"
#define MY_OBJECT    "AppleTalk-Server"

WSH_REGISTER_NAME    atname;
SOCKADDR_AT          ataddr;
SOCKET               s;
//
// Fill in the name to register
//
strcpy(atname.ObjectName, MY_OBJECT);
atname.ObjectNameLen = strlen(MY_OBJECT);
strcpy(atname.TypeName, MY_TYPE);
atname.TypeNameLen = strlen(MY_TYPE);
strcpy(atname.ZoneName, MY_ZONE);
atname.ZoneNameLen = strlen(MY_ZONE);

s = socket(AF_APPLETALK, SOCK_STREAM, ATPROTO_ADSP);
if (s == INVALID_SOCKET)
{
    // Error
}
ataddr.sat_socket = 0;
ataddr.sat_family = AF_APPLETALK;
if (bind(s, (SOCKADDR *)&ataddr, sizeof(ataddr)) == SOCKET_ERROR)
{
    // Unable to open an endpoint on the AppleTalk network
}
if (setsockopt(s, SOL_APPLETALK, SO_REGISTER_NAME,
               (char *)&atname, sizeof(WSH_NBP_NAME)) == SOCKET_ERROR)
{
    // Name registration failed!
}
```

The first thing you'll notice is the *MY_ZONE*, *MY_TYPE*, and *MY_OBJECT* strings. Remember that an AppleTalk name is three-tiered. Notice that the zone is an asterisk (*). This is a special character used in the zone field to specify the "current" zone the computer is located in. Next we create a socket of type *SOCK_STREAM* of the AppleTalk protocol ADSP. Following socket creation, you'll notice a call to the *bind* function with an address structure that has a zeroed-out *sat_socket* field and only the protocol family field set. This is important because it creates an endpoint on the AppleTalk network for your application to make requests from. Note that while this call to *bind* allows you to perform simple actions on the network, it doesn't by itself allow your application to accept incoming connection requests from clients. To accept client connections, you must register your name on the network, which is the next step.

Registering an AppleTalk name is simple. Make the call to *setsockopt* by passing *SOL_APPLETALK* as the *level* parameter and *SO_REGISTER_NAME* as the *optname* parameter. The last two parameters are a pointer to our *WSH_REGISTER_NAME* structure and its size. If the call succeeds, our server name was successfully registered. If the call fails, the requested name is probably already in use by someone else. The Winsock error returned is *WSAEADDRINUSE* (10048 or 0x02740h). Note that for both datagram-oriented and stream-oriented AppleTalk protocols, a process that wants to receive data must register a name that clients can either send datagrams to or connect to.

Resolving an AppleTalk name

On the client side of the equation, an application usually knows a server by its friendly name and must resolve that into the network, node, and socket numbers used by Winsock calls. This is accomplished by calling *getsockopt* with the *SO_LOOKUP_NAME* option. Performing a name lookup relies on the *WSH_LOOKUP_NAME* structure. This structure and its dependent structure are defined as

```
typedef struct
{
    WSH_ATALK_ADDRESS   Address;
    USHORT              Enumerator;
    WSH_NBP_NAME        NbpName;
} WSH_NBP_TUPLE, *PWSH_NBP_TUPLE;

typedef struct _WSH_LOOKUP_NAME
{
    //  Array of NoTuple WSH_NBP_TUPLEs
    WSH_NBP_TUPLE       LookupTuple;
    ULONG               NoTuples;
} WSH_LOOKUP_NAME, *PWSH_LOOKUP_NAME;
```

When we call *getsockopt* with the *SO_LOOKUP_NAME* option, we pass a buffer cast as a *WSH_LOOKUP_NAME* structure and fill in the *WSH_NBP_NAME* field within the first *LookupTuple* member. Upon a successful call, *getsockopt* returns an array of *WSH_NBP_TUPLE* elements containing physical address information for that name. Figure 6-1 contains the file Atalknm.c, which illustrates how to look up a name. In addition, it shows how to list all "discovered" AppleTalk zones and how to find your default zone. Zone information can be obtained by using the *getsockopt* options *SO_LOOKUP_ZONES* and *SO_LOOKUP_MYZONE*.

Figure 6-1. *AppleTalk name and zone lookup*

```
#include <winsock.h>
#include <atalkwsh.h>

#include <stdio.h>
#include <stdlib.h>
```

```
#define DEFAULT_ZONE       "*"
#define DEFAULT_TYPE       "Windows Sockets"
#define DEFAULT_OBJECT     "AppleTalk-Server"

char szZone[MAX_ENTITY],
     szType[MAX_ENTITY],
     szObject[MAX_ENTITY];

BOOL bFindName = FALSE,
     bListZones = FALSE,
     bListMyZone = FALSE;

void usage()
{
    printf("usage: atlookup [options]\n");
    printf("         Name Lookup:\n");
    printf("           -z:ZONE-NAME\n");
    printf("           -t:TYPE-NAME\n");
    printf("           -o:OBJECT-NAME\n");
    printf("         List All Zones:\n");
    printf("           -lz\n");
    printf("         List My Zone:\n");
    printf("           -lm\n");
    ExitProcess(1);
}

void ValidateArgs(int argc, char **argv)
{
    int               i;

    strcpy(szZone, DEFAULT_ZONE);
    strcpy(szType, DEFAULT_TYPE);
    strcpy(szObject, DEFAULT_OBJECT);

    for(i = 1; i < argc; i++)
    {
        if (strlen(argv[i]) < 2)
            continue;
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'z':        // Specify a zone name
                    if (strlen(argv[i]) > 3)
                        strncpy(szZone, &argv[i][3], MAX_ENTITY);
                    bFindName = TRUE;
                    break;
                case 't':        // Specify a type name
                    if (strlen(argv[i]) > 3)
                        strncpy(szType, &argv[i][3], MAX_ENTITY);
                    bFindName = TRUE;
                    break;
                case 'o':        // Specify an object name
```

```
                    if (strlen(argv[i]) > 3)
                        strncpy(szObject, &argv[i][3], MAX_ENTITY);
                    bFindName = TRUE;
                    break;
                case 'l':        // List zones information
                    if (strlen(argv[i]) == 3)
                        // List all zones
                        if (tolower(argv[i][2]) == 'z')
                            bListZones = TRUE;
                        // List my zone
                        else if (tolower(argv[i][2]) == 'm')
                            bListMyZone = TRUE;
                    break;
                default:
                    usage();
            }
        }
    }
}

int main(int argc, char **argv)
{
    WSADATA              wsd;
    char                 cLookupBuffer[16000],
                        *pTupleBuffer = NULL;
    PWSH_NBP_TUPLE       pTuples = NULL;
    PWSH_LOOKUP_NAME     atlookup;
    PWSH_LOOKUP_ZONES    zonelookup;
    SOCKET               s;
    DWORD                dwSize = sizeof(cLookupBuffer);
    SOCKADDR_AT          ataddr;
    int                  i;

    // Load the Winsock library
    //
    if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
    {
        printf("Unable to load Winsock library!\n");
        return 1;
    }

    ValidateArgs(argc, argv);

    atlookup = (PWSH_LOOKUP_NAME)cLookupBuffer;
    zonelookup = (PWSH_LOOKUP_ZONES)cLookupBuffer;
    if (bFindName)


    {
        // Fill in the name to look up
        //
        strcpy(atlookup->LookupTuple.NbpName.ObjectName, szObject);
        atlookup->LookupTuple.NbpName.ObjectNameLen =
```

```
            strlen(szObject);
        strcpy(atlookup->LookupTuple.NbpName.TypeName, szType);
        atlookup->LookupTuple.NbpName.TypeNameLen = strlen(szType);
        strcpy(atlookup->LookupTuple.NbpName.ZoneName, szZone);
        atlookup->LookupTuple.NbpName.ZoneNameLen = strlen(szZone);
    }
    // Create the AppleTalk socket
    //
    s = socket(AF_APPLETALK, SOCK_STREAM, ATPROTO_ADSP);
    if (s == INVALID_SOCKET)
    {
        printf("socket() failed: %d\n", WSAGetLastError());
        return 1;
    }
    // We need to bind in order to create an endpoint on the
    // AppleTalk network to make our query from
    //
    ZeroMemory(&ataddr, sizeof(ataddr));
    ataddr.sat_family = AF_APPLETALK;
    ataddr.sat_socket = 0;
    if (bind(s, (SOCKADDR *)&ataddr, sizeof(ataddr)) ==
            INVALID_SOCKET)
    {
        printf("bind() failed: %d\n", WSAGetLastError());
        return 1;
    }

    if (bFindName)
    {
        printf("Looking up: %s:%s@%s\n", szObject, szType, szZone);
        if (getsockopt(s, SOL_APPLETALK, SO_LOOKUP_NAME,
                    (char *)atlookup, &dwSize) == INVALID_SOCKET)
        {
            printf("getsockopt(SO_LOOKUP_NAME) failed: %d\n",
                WSAGetLastError());
            return 1;
        }
        printf("Lookup returned: %d entries\n",
            atlookup->NoTuples);
        //
        // Our character buffer now contains an array of
        // WSH_NBP_TUPLE structures after our WSH_LOOKUP_NAME
        // structure
        //
        pTupleBuffer = (char *)cLookupBuffer +
            sizeof(WSH_LOOKUP_NAME);
        pTuples = (PWSH_NBP_TUPLE) pTupleBuffer;

        for(i = 0; i < atlookup->NoTuples; i++)
        {
            ataddr.sat_family = AF_APPLETALK;
            ataddr.sat_net    = pTuples[i].Address.Network;
            ataddr.sat_node   = pTuples[i].Address.Node;
```

```
            ataddr.sat_socket = pTuples[i].Address.Socket;
            printf("server address = %lx.%lx.%lx.\n",
                ataddr.sat_net,
                ataddr.sat_node,
                ataddr.sat_socket);
        }
    }
    else if (bListZones)
    {
        // It is very important to pass a sufficiently big buffer
        // for this option. Windows NT 4 SP3 blue screens if it
        // is too small.
        //
        if (getsockopt(s, SOL_APPLETALK, SO_LOOKUP_ZONES,
                  (char *)atlookup, &dwSize) == INVALID_SOCKET)
        {
            printf("getsockopt(SO_LOOKUP_NAME) failed: %d\n",
                WSAGetLastError());
            return 1;
        }
        printf("Lookup returned: %d zones\n", zonelookup->NoZones);
        //
        // The character buffer contains a list of null-separated
        //  strings after the WSH_LOOKUP_ZONES structure
        //
        pTupleBuffer = (char *)cLookupBuffer +
            sizeof(WSH_LOOKUP_ZONES);
        for(i = 0; i < zonelookup->NoZones; i++)
        {
            printf("%3d: '%s'\n", i+1, pTupleBuffer);
            while (*pTupleBuffer++);
        }
    }
    else if (bListMyZone)
    {
        // This option returns a simple string
        //



        if (getsockopt(s, SOL_APPLETALK, SO_LOOKUP_MYZONE,
                 (char *)cLookupBuffer, &dwSize) == INVALID_SOCKET)
        {
            printf("getsockopt(SO_LOOKUP_NAME) failed: %d\n",
                WSAGetLastError());
            return 1;
        }
        printf("My Zone: '%s'\n", cLookupBuffer);
    }
    else
        usage();

    WSACleanup();
```

```
    return 0;
}
```

When you are using most of the AppleTalk socket options—such as *SO_LOOKUP_MYZONE*, *SO_LOOKUP_ZONES*, and *SO_LOOKUP_NAME*—you need to provide a large character buffer to the *getsockopt* call. If you call an option that requires you to provide a structure, that structure needs to be at the start of the supplied character buffer. If the call to *getsockopt* is successful, the function places return data in the character buffer after the end of the supplied structure. Take a look at the *SO_LOOKUP_NAME* section in Figure 6-1. The variable *cLookupBuffer* is a simple character array used in the call to *getsockopt*. First we cast it as a *PWSH_LOOKUP_NAME* and fill in the name information we want to find. We pass the buffer into *getsockopt*, and upon return, we increment the character pointer *pTupleBuffer* so that it points to the character after the end of the *WSH_LOOKUP_NAME* structure. Next we cast that pointer to a variable of *PWSH_NBP_TUPLE* because the data returned from a lookup name call is an array of *WSH_NBP_TUPLE* structures. Once we have the proper starting location and type of the tuples, we can walk through the array. Chapter 9 contains more in-depth information about the various socket options specific to the AppleTalk address family.

Creating a Socket

AppleTalk is available in Winsock 1.1 and later, so you can use either socket-creation routine. Again, you have two options of how to specify the underlying AppleTalk protocols. First you can supply the corresponding define from *Atalkwsh.h* for the protocol you want, or you can enumerate the protocols using *WSAEnumProtocols* and passing the *WSAPROTOCOL_INFO* structure. Table 6-1 lists the required parameters for each AppleTalk protocol type when you create a socket directly using *socket* or *WSASocket*.

Table 6-1. *AppleTalk protocols and parameters*

Protocol	Address Family	Socket Type	Protocol Type
MSAFD AppleTalk [ADSP]		*SOCK_RDM*	*ATPROTO_ADSP*
MSAFD AppleTalk [ADSP] [Pseudo-Stream]		*SOCK_STREAM*	*ATPROTO_ADSP*
	AF_APPLETALK		
MSAFD AppleTalk [PAP]		*SOCK_RDM*	*ATPROTO_PAP*
MSAFD AppleTalk [RTMP]		*SOCK_DGRAM*	*DDPPROTO_RTMP*
MSAFD AppleTalk [ZIP]		*SOCK_DGRAM*	*DDPPROTO_ZIP*

ATM

The Asynchronous Transfer Mode (ATM) protocol is one of the newest protocols available that is supported by Winsock 2 on Windows 98 and Windows 2000. ATM is usually used for high-speed networking on LANs and WANs and can be used for all types of communication, such as voice, video, and data requiring high-speed communication. In general, ATM provides guaranteed quality of service (QOS) using Virtual Connections (VCs) on a network. As you will see in a moment, Winsock is capable of using VCs on an ATM network through the ATM address family. An ATM network—as shown in Figure 6-2—typically comprises endpoints (or computers) that are interconnected by switches that bridge an ATM network together.

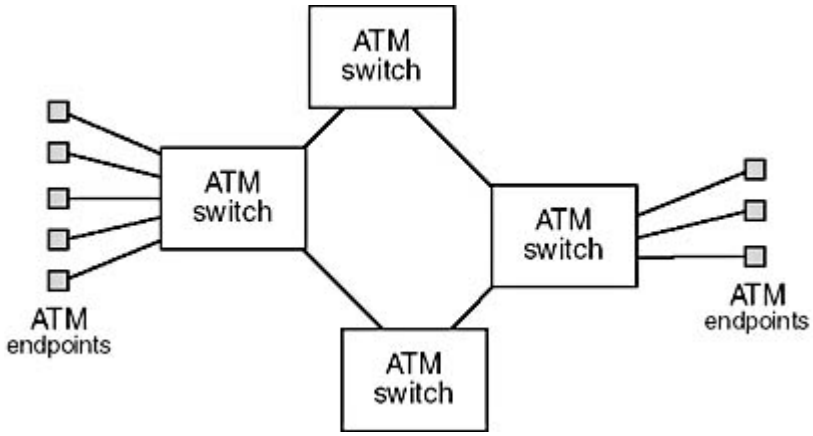

Figure 6-2. *ATM network*

There are a few things to be aware of when programming for the ATM protocol. First, ATM is a media type and not really a protocol per se. That is, ATM is similar to writing Ethernet frames directly on an Ethernet network. Like Ethernet, the ATM protocol doesn't provide flow control. It is a connection-oriented protocol that provides either message or stream modes. This also means that a sending application might overrun the local buffers if data cannot be sent quickly enough. Likewise, a receiving application must post receives frequently; otherwise, when the receiving buffers become full, any additional incoming data might be dropped. If your application requires flow control, one alternative is to use IP over ATM, which is simply the IP protocol running over an ATM network. As a result, the application follows the IP address family described above. Of course, ATM does offer some advantages over IP, such as a rooted multicast scheme (described in Chapter 12); however, the protocol that best suits you depends on your application's needs.

> NOTE
>
> Since ATM is new to Winsock 2, the information in this section was tested against ATM implementations on Windows 2000 Beta 3 only. Windows 98 (Service Pack 1) wasn't available for testing at the time of this writing, and it is possible that some of the information might not conform to the final implementation details of both Windows 2000 and Windows 98 (Service Pack 1).

Addressing

An ATM network has two network interfaces: the user network interface (UNI) and the network node interface (NNI). The UNI interface is the communication established between an endpoint and an ATM switch, while the NNI interface is the communication established between two switches. Each of these interfaces has a related communication protocol, described below.

- UNI signaling protocol Allows an endpoint to establish communication on an ATM network by sending setup and control information between an endpoint and an ATM switch. Note that this protocol is limited to

transmissions between an endpoint and an ATM switch and isn't directly transmitted over an ATM network through switches.

- NNI signaling protocol Allows an ATM switch to communicate routing and control information between two switches.

For purposes of setting up an ATM connection through Winsock, we will only discuss certain information elements in the UNI signaling protocol. Winsock on Windows 2000 and Windows 98 (service pack 1) currently supports the UNI version 3.1 signaling protocol.

Winsock allows a client/server application to communicate over an ATM network by setting up a Service Access Point (SAP) to form connections using the ATM UNI signaling protocol. ATM is a connection-oriented protocol that requires endpoints to establish virtual connections across an ATM network for communication. An SAP simply allows Winsock applications to register and identify a socket interface for communication on an ATM network through a *SOCKADDR_ATM* address structure. Once an SAP is established, Winsock uses the SAP to establish a virtual connection between a Winsock client and server over ATM by making calls to the ATM network using the UNI signaling protocol. The *SOCKADDR_ATM* structure is defined as

```
typedef struct sockaddr_atm
{
    u_short      satm_family;
    ATM_ADDRESS  satm_number;
    ATM_BLLI     satm_blli;
    ATM_BHLI     satm_bhli;
} sockaddr_atm, SOCKADDR_ATM, *PSOCKADDR_ATM, *LPSOCKADDR_ATM;
```

The *satm_family* field should always be set to *AF_ATM*. The *satm_number* field represents an actual ATM address represented as an *ATM_ADDRESS* structure using one of two basic ATM addressing schemes: E.164 and Network Service Access Point (NSAP). NSAP addresses are also referred to as an NSAP-style ATM Endsystem Address (AESA). The *ATM_ADDRESS* structure is defined as

```
typedef struct
{
    DWORD AddressType;
    DWORD NumofDigits;
    UCHAR Addr[ATM_ADDR_SIZE];
} ATM_ADDRESS;
```

The *AddressType* field defines the specified addressing scheme. This should be set to *ATM_E164* for the E.164 addressing scheme and *ATM_NSAP* for the NSAP-style addressing scheme. Additionally, the *AddressType* field can be set to other values defined in Table 6-2 on the following when an application tries to bind a socket to an SAP, which we will discuss in more detail later in this chapter. The *NumofDigits* field should always be set to *ATM_ADDR_SIZE*. The *Addr* field represents an actual ATM 20-byte E.164 or NSAP address.

The *satm_blli* and *satm_bhli* fields of the *SOCKADDR_ATM* structure represent Broadband Lower Layer Information (BLLI) and Broadband Higher Layer Information (BHLI) in ATM UNI signaling, respectively. In general, these structures are used to identify the protocol stack that operates over an ATM connection. Several well-known combinations of BHLI and BLLI values are described in ATM Form/IETF documents. (A particular combination of values identifies a connection as being used by LAN Emulation over ATM, another combination identifies native IP over ATM, and so on.) Complete ranges of values for the fields in these structures are given in the ATM UNI 3.1 standards book. ATM Form/IETF documents can be found at http://www.ietf.org.

Table 6-2. *ATM socket address types*

ATM_ADDRESS AddressType	Setting Type of Address
ATM_E164	An E.164 address; applies when connecting to an SAP
ATM_NSAP	An NSAP-style ATM Endsystem Address (AESA); applies when connecting to an SAP
SAP_FIELD_ANY_AESA_SEL	An NSAP-style ATM Endsystem Address with the selector octet wildcarded; applies to binding a socket to an SAP
SAP_FIELD_ANY_AESA_REST	An NSAP-style ATM Endsystem Address with all the octets except for the selector octet wildcarded; applies to binding a socket to an SAP

The BHLI and BLLI data structures are defined as

```
typedef struct
{
    DWORD HighLayerInfoType;
    DWORD HighLayerInfoLength;
    UCHAR HighLayerInfo[8];
} ATM_BHLI;

typedef struct
{
    DWORD Layer2Protocol;
    DWORD Layer2UserSpecifiedProtocol;
    DWORD Layer3Protocol;
    DWORD Layer3UserSpecifiedProtocol;
    DWORD Layer3IPI;
    UCHAR SnapID[5];
} ATM_BLLI;
```

Further details of the definition and use of these fields are beyond the scope of this book. An application that simply wants to form Winsock communication over an ATM network should set the following fields in the BHLI and BLLI structures to the *SAP_FIELD_ABSENT* value:

- ATM_BLLI—Layer2Protocol
- ATM_BLLI—Layer3Protocol
- ATM_BHLI—HighLayerInfoType

When these fields are set to this value, none of the other fields in both structures are used. The following pseudocode demonstrates how an application might use the *SOCKADDR_ATM* structure to set up an SAP for an NSAP address:

```
SOCKADDR_ATM atm_addr;
UCHAR MyAddress[ATM_ADDR_SIZE];

atm_addr.satm_family                 = AF_ATM;
atm_addr.satm_number.AddressType     = ATM_NSAP;
atm_addr.satm_number.NumofDigits     = ATM_ADDR_SIZE;
atm_addr.satm_blli.Layer2Protocol    = SAP_FIELD_ABSENT;
atm_addr.satm_blli.Layer3Protocol    = SAP_FIELD_ABSENT;
atm_addr.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;

memcpy(&atm_addr.satm_number.Addr, MyAddress, ATM_ADDR_SIZE);
```

ATM addresses are normally represented as a hexadecimal ASCII string of 40 characters, which corresponds to the 20 bytes that make up either an NSAP-style or an E.164 address in an *ATM_ADDRESS* structure. For example, an ATM NSAP-style address might look like this:

```
47000580FFE1000000F21A1D540000D10FED5800
```

Converting this string to a 20-byte address can be a rather tedious task. However, Winsock provides a protocol-independent API function, *WSAStringToAddress*, which allows you to convert a 40-character ATM hexadecimal ASCII string to an *ATM_ADDRESS* structure. We describe this API function in more detail at the end of this chapter. Another way to convert a hexadecimal ASCII string to hexadecimal (binary) format is to use the function *AtoH* defined in Figure 6-3. This function isn't a part of Winsock. However, it is simple enough to develop, and you will see it in the samples in Chapter 7.

Figure 6-3. *Conversion functions for ATM hexadecimal strings*

```
//
// Function: AtoH
//
// Description: This function coverts the ATM
// address specified in string (ASCII) format to
// binary (hexadecimal) format
//
void AtoH(CHAR *szDest, CHAR *szSource, INT iCount)
{
    while (iCount--)
    {
        *szDest++ = ( BtoH ( *szSource++ ) << 4 )
            + BtoH ( *szSource++ );
    }
    return;
}
//
// Function: BtoH

//
// Description: This function returns the equivalent
// binary value for an individual character specified
```

```
// in ASCII format
//
UCHAR BtoH( CHAR ch )
{
    if ( ch >= '0' && ch <= '9' )
    {
        return ( ch - '0' );
    }

    if ( ch >= 'A' && ch <= 'F' )
    {
        return ( ch - 'A' + 0xA );
    }

    if (ch >= 'a' && ch <= 'f' )
    {
        return ( ch - 'a' + 0xA );
    }
    //
    // Illegal values in the address will not be
    // accepted
    //
    return -1;
}
```

Creating a Socket

In ATM, applications can create only connection-oriented sockets because ATM allows communication only over a VC. Therefore, data can be transmitted either as a stream of bytes or in a message-oriented fashion. To open a socket using the ATM protocol, call the *socket* function or the *WSASocket* function with the address family *AF_ATM* and the socket type *SOCK_RAW*, and set the protocol field to *ATMPROTO_AAL5*. For example:

```
s = socket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5);

s = WSASocket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5, NULL, 0,
    WSA_FLAG_OVERLAPPED);
```

By default, opening a socket (as in the example) creates a stream-oriented ATM socket. Windows also features an ATM provider that can perform message-oriented data transfers. Using the message-oriented provider requires you to explicitly specify the native ATM protocol provider to the *WSASocket* function by using a *WSAPROTOCOL_INFO* structure, as described in Chapter 5. This is necessary because the three elements in the *socket* call and the *WSASocket* call (address family, socket type, and protocol) match every ATM provider available in Winsock. By default, Winsock returns the protocol entry that matches those three attributes and that is marked as default, which in this case is the stream-oriented provider. The following pseudocode demonstrates how to retrieve the ATM message-oriented provider and establish a socket:

```
dwRet = WSAEnumProtocols(NULL, lpProtocolBuf, &dwBufLen);

for (i = 0; i < dwRet; i++)
{
    if ((lpProtocolBuf[i].iAddressFamily == AF_ATM) &&
       (lpProtocolBuf[i].iSocketType == SOCK_RAW) &&
       (lpProtocolBuf[i].iProtocol == ATMPROTO_AAL5) &&
       (lpProtocolBuf[i].dwServiceFlags1 &
         XP1_MESSAGE_ORIENTED))
    {
        s = WSASocket(FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
           FROM_PROTOCOL_INFO, lpProtocolBuf[i], 0,
           WSA_FLAG_OVERLAPPED);
    }
}
```

Binding a Socket to an SAP

ATM addresses are actually quite complicated because the 20 bytes they comprise contain many informational elements. Winsock application programmers need not worry about all the specific details of these elements with the exception of the last byte. The last byte in NSAP-style and E.164 addresses represents a selector value that uniquely allows your application to define and specify a particular SAP on an endpoint. As we described earlier, Winsock uses an SAP to form communication over an ATM network.

When Winsock applications want to communicate over ATM, a server application must register an SAP on an endpoint and wait for a client application to connect on the registered SAP. For a client application, this simply involves setting up a *SOCKADDR_ATM* structure with the *ATM_E164* or *ATM_NSAP* address type and supplying the ATM address associated with the server's SAP. To create an SAP to listen for connections, your application must first create a socket for the *AF_ATM* address family. Once the socket is created, your application must define a *SOCKADDR_ATM* structure using the *SAP_FIELD_ANY_AESA_SEL*, *SAP_FIELD_ANY_AESA_REST*, *ATM_E164*, or *ATM_NSAP* address type as defined in Table 6-2 on 168. For an ATM socket, an SAP will be created once your application calls the Winsock *bind* API function (which we describe in Chapter 7), and these address types define how Winsock creates an SAP on your endpoint.

The address type *SAP_FIELD_ANY_AESA_SEL* tells Winsock to create an SAP that is capable of listening for any incoming ATM Winsock connection, which is known as wildcarding an ATM address and the selector. This means that only one socket can be bound to this endpoint listening for any connection—if another socket tries to bind with this address type, it will fail with Winsock error *WSAEADDRINUSE*. However, you can have another socket bound explicitly to your endpoint on a particular selector. The address type *SAP_FIELD_ANY_AESA_REST* can be used to create an SAP that is explicitly bound to a specified selector on an endpoint. This is known as wildcarding only the ATM address and not the selector. You can have only one socket at a time bound to a particular selector on an endpoint, or the *bind* call will fail with error *WSAEADDRINUSE*. When you use the *SAP_FIELD_ANY_AESA_SEL* type, you should specify an ATM address of all zeros in the *ATM_ADDRESS* structure. If you use *SAP_FIELD_ANY_AESA_REST*, you should specify all zeros for the first 19 bytes of the ATM address and the last byte should indicate what selector number you plan to use.

Sockets that are bound to explicit selectors (*SAP_FIELD_ANY_AESA_REST*) take higher precedence than those sockets that are bound to a wildcarded selector (*SAP_FIELD_ANY_AESA_SEL*). Those sockets that are bound to explicit selectors (*SAP_FIELD_ANY_AESA_REST*) or explicit interfaces (*ATM_NSAP* and *ATM_E164*) will get first choice at connections. (That is, if a connection comes in on the endpoint and the selector that a socket is explicitly listening on, that socket gets the connection.) Only when no explicitly bound socket is available will a wildcarded selector socket get the connection. Chapter 7 further demonstrates how to set up a socket that listens for connections on an SAP.

Finally, a utility named Atmadm.exe allows you to retrieve all ATM address and virtual connection information on an endpoint. This utility can be useful when you are developing an ATM application and need to know which interfaces are available on an endpoint. The command line options listed in the following table are available.

Parameter	Description
-c	List all connections (VC). Lists the remote address and the local interface.
-a	Lists all registered addresses (i.e., all local ATM interfaces and their addresses).
-s	Prints statistics (current number of calls, number of signaling and ILMI packets sent/received, etc.).

Name Resolution

Currently no name providers are available for ATM under Winsock. This unfortunately requires applications to specify the 20-byte address ATM to establish socket communication over an ATM network. Chapter 10 discusses the Windows 2000 domain name space that can be generically used to register ATM addresses with user-friendly service names.

Additional Winsock 2 Support Functions

Winsock 2 provides two useful support functions named *WSAAddressToString* and *WSAStringToAddress* that provide a protocol-independent method to convert a *SOCKADDR* structure of a protocol to a formatted character string and vice versa. Since these functions are protocol-independent, they require the transport protocol to support the string conversions. Currently these functions work only for the *AF_INET* and *AF_ATM* address families. The *WSAAddressToString* function is defined as

```
INT WSAAddressToString(
    LPSOCKADDR lpsaAddress,
    DWORD dwAddressLength,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    OUT LPTSTR lpszAddressString,
    IN OUT LPDWORD lpdwAddressStringLength
);
```

The *lpsaAddress* parameter represents a *SOCKADDR* structure for a particular protocol that contains the address to convert to a string. The *dwAddressLength* parameter specifies the size of the first parameter's structure, which can vary in size with different protocols. The *lpProtocolInfo* is an optional parameter that represents a protocol provider. Protocol providers can be retrieved from the *WSAEnumProtocols* API function, as described in Chapter 5. If you specify *NULL*, the call uses the provider of the first protocol supporting the address family indicated in *lpsaAddress.* The *lpszAddressString* parameter is a buffer that receives the human-readable address string. The *lpdwAddressStringLength* parameter represents the size of *lpszAddressString*. On output, it returns the length of the string actually copied into *lpszAddressString*. If the supplied buffer isn't large enough, the function fails with error *WSAEFAULT* and the *lpdwAddressStringLength* parameter is updated with the required size in bytes.

Conversely, the *WSAStringToAddress* API function takes a human-readable address string and converts it to a *SOCKADDR* structure. *WSAStringToAddress* is defined as

```
INT WSAStringToAddress(
    LPTSTR AddressString,
    INT AddressFamily,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    LPSOCKADDR lpAddress,
    LPINT lpAddressLength
);
```

The *AddressString* parameter is a human-readable address string. Table 6-3 describes the format for this string for the current supported protocols.

Table 6-3. *Address string formats*

Address Family	String Format
IP	*XXX.XXX.XXX.XXX*:*Y*—where *X* represents an octet in an IP address string and *Y* represents a port number
ATM	*NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN*— where the 40 *N* characters represent a 20-byte ATM address in hexadecimal notation

The *AddressFamily* parameter represents the address family type for the *AddressString* parameter. The *lpProtocolInfo* parameter is an optional parameter that represents a protocol provider. If you set this parameter to *NULL*, Winsock will search for the first available protocol provider for the address family type specified in the

AddressFamily parameter. If you want to select a particular provider, the *WSAEnumProtocols* API function will supply you with a list of available protocol providers installed on your system. The *Address* buffer parameter takes a *SOCKADDR* structure that receives the information in the address string. The *lpAddressLength* parameter represents the size of the resultant *SOCKADDR* structure.

Conclusion

In this chapter, we described the protocol address families supported by Winsock and explained addressing attributes specific to each family. For each address family, we discussed how to create a socket and how to set up a socket address structure to begin communication over a protocol. The next chapter will describe basic communication techniques available in Winsock, which apply to all of the address families described in this chapter.

Chapter 7

Winsock Basics

This chapter is dedicated to learning the basic techniques and API calls necessary for writing successful network applications. In the last chapter, you learned how each protocol accessible from Winsock addresses machines and services on those machines. In this chapter, we'll look at establishing a connection from one machine on the network to another, along with how to send and receive data. For simplicity's sake, and to prevent repetition, the discussion in this chapter is limited to the TCP/IP protocol. However, this book's companion CD contains client/server samples for each of the protocols covered in Chapter 6. The only protocol-dependent operation is socket creation. Most of the remaining Winsock calls that are required for establishing a connection and for sending and receiving data are independent of the underlying protocol. The exceptions were noted in Chapter 6 along with each protocol discussion.

The examples presented in this chapter help to provide an understanding of the Winsock calls that are required for accepting connections, establishing connections, and sending and receiving data. Because the purpose of this chapter is to learn these Winsock calls, the examples presented use straight blocking Winsock calls. Chapter 8 presents the different I/O models available in Winsock, including code examples.

Additionally, in this chapter we will present both the Winsock 1 and Winsock 2 versions of the various API functions. You can differentiate the two functions with the *WSA* prefix. If Winsock 2 updated or added a new API function in its specification, the function name is prefixed with *WSA*. For example, the Winsock 1 function to create a socket is simply *socket*. Winsock 2 introduces a newer version named *WSASocket* that is capable of using some of the new features made available in Winsock 2. There are a few exceptions to this naming rule. *WSAStartup*, *WSACleanup*, *WSARecvEx*, and *WSAGetLastError* are in the Winsock 1.1 specification.

Initializing Winsock

Every Winsock application must load the appropriate version of the Winsock DLL. If you fail to load the Winsock library before calling a Winsock function, the function will return a *SOCKET_ERROR* and the error will be *WSANOTINITIALISED*. Loading the Winsock library is accomplished by calling the *WSAStartup* function, which is defined as

```
int WSAStartup(
    WORD  wVersionRequested,
    LPWSADATA  lpWSAData
);
```

The *wVersionRequested* parameter is used to specify the version of the Winsock library you want to load. The high-order byte specifies the minor version of the requested Winsock library, while the low-order byte is the major version. You can use the handy macro *MAKEWORD(x, y)*, in which *x* is the high byte and *y* is the low byte, to obtain the correct value for *wVersionRequested*.

The *lpWSAData* parameter is a pointer to a *LPWSADATA* structure that *WSAStartup* fills with information related to the version of the library it loads:

```
typedef struct WSAData
{
    WORD           wVersion;
    WORD           wHighVersion;
    char           szDescription[WSADESCRIPTION_LEN + 1];
    char           szSystemStatus[WSASYS_STATUS_LEN + 1];
    unsigned short iMaxSockets;
    unsigned short iMaxUdpDg;
    char FAR *     lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

WSAStartup sets the first field, *wVersion*, to the Winsock version you will be using. The *wHighVersion* parameter holds the highest version of the Winsock library available. Remember that in both of these fields, the high-order byte represents the Winsock minor version, while the low-order byte is the major version. The *szDescription* and *szSystemStatus* fields are set by the particular implementation of Winsock and aren't really useful. Do not use the next two fields, *iMaxSockets* and *iMaxUdpDg*. They are supposed to be the maximum number of concurrently open sockets and the maximum datagram size; however, to find the maximum datagram size you should query the protocol information through *WSAEnumProtocols*. The maximum number of concurrent sockets isn't some magic number—it depends more on how much physical memory is available. Finally, the *lpVendorInfo* field is reserved for vendor-specific information regarding the implementation of Winsock. This field is not used on any Win32 platforms.

Table 7-1 lists the latest versions of Winsock that the various Microsoft Windows platforms support. What's important to remember is the difference between major versions. Winsock 1.x does not support many of the advanced Winsock features detailed in this section. Additionally, for applications using Winsock 1, the include file Winsock.h is necessary; otherwise, for Winsock 2, Winsock2.h should be included.

Table 7-1. *Supported Winsock versions*

Platform	*Winsock Version*
Windows 95	1.1 (2.2)
Windows 98	2.2
Windows NT 4.0	2.2
Windows 2000	2.2
Windows CE	1.1

> NOTE
>
> A Winsock 2 upgrade for Windows 95 is available for download from http://www.microsoft.com/windows95/downloads/.

Note that even though a platform supports Winsock 2, you do not have to request the latest version. That is, if you want to write an application that is supported on a majority of platforms, you should write it to the Winsock 1.1 specification. This application will run perfectly well on Windows NT 4.0 because all Winsock 1.1 calls are mapped through the Winsock 2 DLL. Also, if a newer version of the Winsock library becomes available for a platform that you use, it is often in your best interest to upgrade. These new versions contain bug fixes, and your old code should run without a problem—at least theoretically. In some cases, the behavior of the Winsock stack is different from what the specification defines. As a result, many programmers write their applications according to the behavior of the particular platform they are targeting instead of the specification. For example, under Windows NT 4.0, when a program is using the asynchronous window event model, an *FD_WRITE* is posted after every successful *send* or *WSASend* to indicate that you can write data. However, the specification says that an *FD_WRITE* is posted when the system is able to send data, such as when the application starts, and that a posted *FD_WRITE* means you should keep writing until you receive the error *WSAEWOULDBLOCK*. In fact, after the system sends all pending data and can process more *send* and *WSASend* calls, it will post an *FD_WRITE* event to your application window, at which time you can resume writing data to the network (Knowledge Base Article Q186245). This problem has been fixed in Service Pack 4 for Windows NT 4.0 as well as in Windows 2000.

For the most part, however, when writing new applications you will load the latest version of the Winsock library currently available. Remember that if, for example, Winsock 3 is released, your application that loads version 2.2 should run as expected. If you request a Winsock version later than that which the platform supports, *WSAStartup* will fail. Upon return, the *wHighVersion* of the *WSADATA* structure will be the latest version supported by the library on the current system.

Error Checking and Handling

We'll first cover error checking and error handling, as they are vital to writing a successful Winsock application. It is actually common for Winsock functions to return an error; however, many times the error is not critical and communication can still take place on that socket. The most common return value for an unsuccessful Winsock call is *SOCKET_ERROR*, although this is certainly not always the case. When covering each API call in detail, we'll point out the return value corresponding to an error. The constant *SOCKET_ERROR* actually is -1. If you make a call to a Winsock function and an error condition occurs, you can use the function *WSAGetLastError* to obtain a code that indicates specifically what happened. This function is defined as

```
int WSAGetLastError (void);
```

A call to the function after an error occurs will return an integer code for the particular error that occurred. These error codes returned from *WSAGetLastError* all have predefined constant values that are declared in either Winsock.h or Winsock2.h, depending on the version of Winsock. The only difference between the two header files is that Winsock2.h contains more error codes for some of the newer API functions and capabilities introduced in Winsock 2. The constants defined for the various error codes (with *#define* directives) generally begin with *WSAE*.

Connection-Oriented Protocols

In this first section, we'll cover the Winsock functions necessary for both receiving connections and establishing connections. We'll first discuss how to listen for client connections and explore the process for accepting or rejecting a connection. Then we'll describe how to initiate a connection to a server. Finally, we will discuss how data is transferred in a connection session.

Server API Functions

A server is a process that waits for any number of client connections with the purpose of servicing their requests. A server must listen for connections on a well-known name. In TCP/IP, this name is the IP address of the local interface and a port number. Every protocol has a different addressing scheme and therefore a different naming method. The first step in Winsock is to bind a socket of the given protocol to its wellknown name, which is accomplished with the *bind* API call. The next step is to put the socket into listening mode, which is performed (appropriately enough) with the *listen* API function. Finally, when a client attempts a connection, the server must accept the connection with either the *accept* or the *WSAAccept* call. In the next few sections, we will discuss each API call that is required for binding and listening and for accepting a client connection. Figure 7-1 illustrates the basic calls a server and a client must perform in order to establish a communication channel.

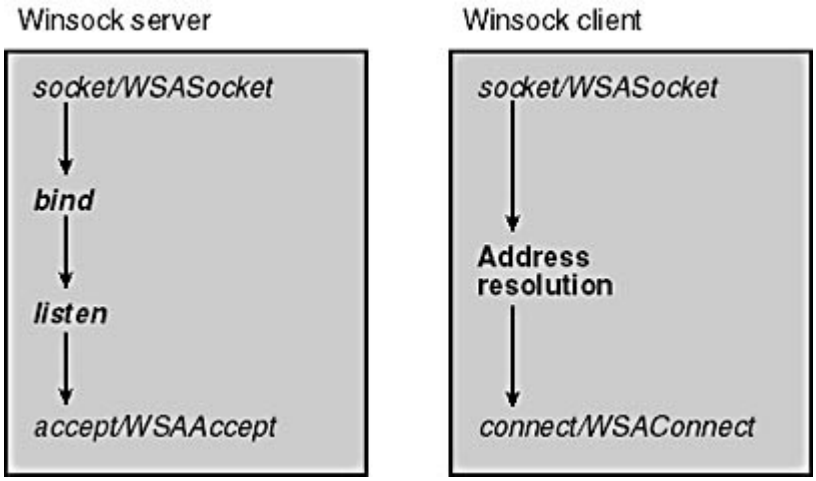

Figure 7-1. *Winsock basics for server and client*

bind

Once the socket of a particular protocol is created, you must bind the socket to a well-known address. The *bind* function associates the given socket with a well-known address. This function is declared as

```
int bind(
     SOCKET                         s,
     const struct sockaddr FAR* name,
     int                            namelen
);
```

The first parameter, *s*, is the socket on which you want to wait for client connections. The second parameter is of type *struct sockaddr*, which is simply a generic buffer. You must actually fill out an address buffer specific to the protocol you are using and cast that as a *struct sockaddr* when calling *bind*. The Winsock header file defines the type *SOCKADDR* as *struct sockaddr*. We'll use this type throughout the chapter for brevity. The third parameter is simply the size of the protocol-specific address structure being passed. For example, the following code illustrates

how this is done on a TCP connection:

```
SOCKET               s;
struct sockaddr_in   tcpaddr;
int                  port = 5150;

s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

tcpaddr.sin_family = AF_INET;
tcpaddr.sin_port = htons(port);
tcpaddr.sin_addr.s_addr = htonl(INADDR_ANY);

bind(s, (SOCKADDR *)&tcpaddr, sizeof(tcpaddr));
```

If the structure *sockaddr_in* looks mysterious to you, consult the TCP/IP addressing section in Chapter 6. From the example, you'll see a stream socket being created, followed by setting up the TCP/IP address structure on which client connections will be accepted. In this case, the socket is being bound to the default IP interface on port number 5150. The call to *bind* formally establishes this association of the socket with the IP interface and port.

On error, *bind* returns *SOCKET_ERROR*. The most common error encountered with *bind* is *WSAEADDRINUSE*. When you're using TCP/IP, the *WSAEADDRINUSE* error indicates that another process is already bound to the local IP interface and port number or that the IP interface and port number are in the *TIME_WAIT* state. If you call *bind* again on a socket that is already bound, *WSAEFAULT* will be returned.

listen

The next piece of the equation is to put the socket into listening mode. The *bind* function merely associates the socket with a given address. The API function that tells a socket to wait for incoming connections is *listen*, which is defined as

```
int listen(
     SOCKET  s,
     int     backlog
);
```

Again, the first parameter is a bound socket. The *backlog* parameter specifies the maximum queue length for pending connections. This is important when several simultaneous requests are made to the server. For example, let's say the backlog parameter is set to 2. If three client requests are made at the same time, the first two will be placed in a "pending" queue so that the application can service their requests. The third connection request will fail with *WSAECONNREFUSED*. Note that once the server accepts a connection, the connection request is removed from the queue so that others can make a request. The *backlog* parameter is silently limited to a value determined by the underlying protocol provider. Illegal values are replaced with their nearest legal values. Additionally, there is no standard provision for finding the actual backlog value.

The errors associated with *listen* are fairly straightforward. By far the most common is *WSAEINVAL,* which usually indicates that you forgot to call *bind* before *listen*. Otherwise, it is possible to receive the *WSAEADDRINUSE* error on the *listen* call as opposed to the *bind* call. This error occurs most often on the *bind* call.

accept and *WSAAccept*

Now you're ready to accept client connections. This is accomplished with either the *accept* or the *WSAAccept* function. The prototype for *accept* is

```
SOCKET accept(
     SOCKET  s,
     struct sockaddr FAR*  addr,
     int FAR*  addrlen
);
```

Parameter *s* is the bound socket that is in a listening state. The second parameter should be the address of a valid *SOCKADDR_IN* structure, while *addrlen* should be a reference to the length of the *SOCKADDR_IN* structure. For a socket of another protocol, substitute the *SOCKADDR_IN* with the *SOCKADDR* structure corresponding to that protocol. A call to *accept* services the first connection request in the queue of pending connections. When the *accept* function returns, the *addr* structure contains the IP address information of the client making the connection request, while the *addrlen* parameter indicates the size of the structure. Additionally, *accept* returns a new socket descriptor that corresponds to the accepted client connection. For all subsequent operations with this client, the new socket should be used. The original listening socket is still used to accept other client connections and is still in listening mode.

Winsock 2 introduced the function *WSAAccept*, which has the ability to conditionally accept a connection based on the return value of a condition function. The prototype for this new function is

```
SOCKET WSAAccept(
     SOCKET  s,
     struct sockaddr FAR *  addr,
     LPINT  addrlen,
     LPCONDITIONPROC  lpfnCondition,
     DWORD  dwCallbackData
);
```

The first three parameters are the same as the Winsock 1 version of *accept*. The *lpfnCondition* argument is a pointer to a function that is called upon a client request. This function determines whether to accept the client's connection request. The prototype for this function is

```
int CALLBACK ConditionFunc(
    LPWSABUF lpCallerId,
    LPWSABUF lpCallerData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
    LPWSABUF lpCalleeId,
    LPWSABUF lpCalleeData,
    GROUP FAR * g,
    DWORD dwCallbackData
);
```

The *lpCallerId* parameter is a value parameter that contains the address of the connecting entity. The *WSABUF* structure is commonly used by many Winsock 2 functions. It is declared as

```
typedef struct __WSABUF {
    u_long      len;
    char FAR * buf;
} WSABUF, FAR * LPWSABUF;
```

Depending on its use, the *len* field refers either to the size of the buffer pointed to by the *buf* field or to the amount of data contained in the data buffer *buf*.

For *lpCallerId*, the *buf* pointer points to an address structure for the given protocol on which the connection is made. To correctly access the information, simply cast the *buf* pointer to the appropriate *SOCKADDR* type. In the case of TCP/IP, this is, of course, a *SOCKADDR_IN* structure that will contain the IP address of the client making the connection. Most network protocols can be expected to support caller ID information at connection-request time.

The *lpCallerData* parameter contains any connection data sent by the client along with the connection request. If caller data was not specified, this parameter is *NULL*. Be aware that most network protocols, such as TCP, do not support connect data. Whether a protocol supports connect or disconnect data can be determined by consulting its entry in the Winsock catalog with the *WSAEnumProtocols* function. See Chapter 5 for the specifics.

The next two parameters, *lpSQOS* and *lpGQOS*, specify any quality of service (QOS) parameters that are being requested by the client. Both parameters reference a QOS structure that contains information regarding bandwidth requirements for both sending and receiving data. If the client is not requesting QOS, these parameters will be *NULL*. The difference between these two parameters is that *lpSQOS* refers to a single connection, while *lpGQOS* is used for socket groups. Socket groups are not implemented or supported in Winsock 1 or 2. (See Chapter 12 for further details about QOS.)

The *lpCalleeId* is another *WSABUF* structure containing the local address to which the client has connected. Again, the *buf* field of this structure points to a *SOCKADDR* object of the appropriate address family. This information is useful in the event that the server is running on a multihomed machine. Remember that if a server binds to the address *INADDR_ANY*, connection requests are serviced on any network interface. This parameter will contain the specific interface on which the connection occurred.

The *lpCalleeData* parameter is the complement of *lpCallerData*. The *lpCalleeData* parameter points to a *WSABUF* structure that the server can use to send data back to the client as a part of the connection request process. If the service provider supports this option, the *len* field indicates the maximum number of bytes the server can send back to the client as a part of this connection request. In this case, the server would copy any number of bytes up to this amount into the *buf* portion of the *WSABUF* structure and update the *len* field to indicate the number of bytes being transferred. If the server does not want to return any connect data, the conditional accept function should set the *len* field to 0 before returning. If the provider does not support connect data, the *len* field will be 0. Again, most protocols do not support data exchange upon accept. In fact, none of the currently supported protocols on any Win32 platform support this feature.

Once the server has processed parameters passed into the conditional function, the server must indicate whether to accept, reject, or defer the client's connection request. If the server is accepting the connection, the conditional function should return *CF_ACCEPT*. Upon rejection, the function should return *CF_REJECT*. If for some reason the decision cannot be made at this time, *CF_DEFER* can be returned. When the server is prepared to handle this connection request, it should call *WSAAccept*. Note that the condition function runs in the same thread as the *WSAAccept* function and should return as soon as possible. Also be aware that for the protocols supported by the current Win32 platforms, the conditional accept function does not imply that the client's connection request is delayed until a value is returned from this conditional function. In most cases, the underlying network stack has already accepted the connection at the time the conditional accept function is called. If the value *CF_REJECT* is returned, the underlying stack simply closes the connection. We won't go into the detailed usage of the conditional acceptance function now, as this information will be more useful in Chapter 12.

If an error occurs, *INVALID_SOCKET* is returned. The most common error encountered is *WSAEWOULDBLOCK* if the listening socket is in asynchronous or nonblocking mode and there is no connection to be accepted. When a conditional function returns *CF_DEFER*, *WSAAccept* returns the error *WSATRY_AGAIN*. If the condition function returns *CF_REJECT*, the *WSAAccept* error is *WSAECONNREFUSED*.

Client API Functions

The client is much simpler and involves fewer steps to set up a successful connection. There are only three steps for a client:

1. Create a socket with *socket* or *WSASocket*.
2. Resolve the server's name (dependent on underlying protocol).
3. Initiate the connection with *connect* or *WSAConnect*.

You already know from Chapter 6 how to create the socket and resolve an IP host name, so the only remaining step is establishing a connection. Chapter 6 also covers the various name-resolution methods for other protocol families.

TCP States

As a Winsock programmer, you are not required to know the actual TCP states, but by knowing them you will gain a better understanding of how the Winsock API calls effect change in the underlying protocol. Additionally, many programmers run into a common problem when closing sockets; the TCP states surrounding a socket closure are of the most interest.

The start state of every socket is the CLOSED state. When a client initiates a connection, it sends a SYN packet to the server and puts the client socket in the SYN_SENT state. When the server receives the SYN packet, it sends a SYN-and-ACK packet, which the client responds to with an ACK packet. At this point, the client's socket is in the ESTABLISHED state. If the server never sends a SYN-ACK packet, the client times out and reverts to the CLOSED state.

When a server's socket is bound and is listening on a local interface and port, the state of the socket is LISTEN. When a client attempts a connection, the server receives a SYN packet and responds with a SYN-ACK packet. The state of the server's socket changes to SYN_RCVD. Finally, the client sends an ACK packet, which causes the state of the server's socket to change to ESTABLISHED.

Once the application is in the ESTABLISHED state, there are two paths for closure. If your application initiates the closure, the closure is known as an active socket closure; otherwise, the socket closure is passive. Figure 7-2 illustrates both an active and a passive closure. If you actively initiate a closure, your application sends a FIN packet. When your application calls *closesocket* or *shutdown*(with *SD_SEND* as its second argument), your application sends a FIN packet to the peer, and the state of your socket changes to FIN_WAIT_1. Normally, the peer responds with an ACK packet, and your socket's state becomes FIN_WAIT_2. If the peer also closes the connection, it sends a FIN packet and your computer responds by sending an ACK packet and placing your socket in the TIME_WAIT state.

The TIME_WAIT state is also called the 2MSL wait state. MSL stands for Maximum Segment Lifetime and represents the amount of time a packet can exist on the network before being discarded. Each IP packet has a time-to-live (TTL) field, which when decremented to 0 causes the packet to be discarded. Each router on the network that handles the packet decrements the TTL by 1 and passes the packet on. Once an application enters the TIME_WAIT state, it remains there for twice the MSL time. This allows TCP to re-send the final ACK in case it's lost, causing the FIN to be retransmitted. After the 2MSL wait state completes, the socket goes to the CLOSED state.

On an active close, two other paths lead to the TIME_WAIT state. In our previous discussion, only one side issues a FIN and receives an ACK response, but the peer is still free to send data until it too closes. This is where the other two paths come into play. In one path—the simultaneous close—a computer and its peer at the other side of a connection issue a close at the same time: the computer sends a FIN packet to the peer and receives a FIN packet from the peer. Then the computer sends an ACK packet in response to the peer's FIN packet and changes its socket to the CLOSING state. Once the computer receives the last ACK packet from the peer, the computer's socket state becomes TIME_WAIT.

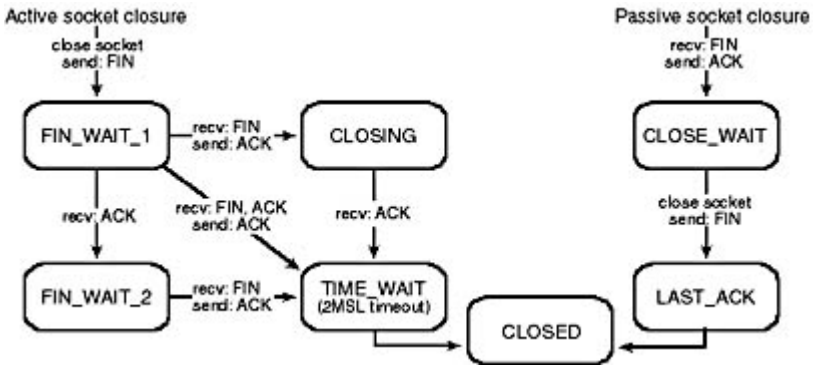

Figure 7-2. *TCP socket closure states*

The other path for an active closure is just a variation on the simultaneous close: the socket transitions from the FIN_WAIT_1 state directly to the TIME_WAIT state. This occurs when an application sends a FIN packet but shortly thereafter receives a FIN-ACK packet from the peer. In this case, the peer is acknowledging the application's FIN packet and sending its own, to which the application responds with an ACK packet.

The major effect of the TIME_WAIT state is that while a TCP connection is in the 2MSL wait state, the socket pair defining that connection cannot be reused. A socket pair is the combination of local IP_local port and remote IPremote port. Some TCP implementations do not allow the reuse of any port number in a socket pair in the TIME_WAIT state. Microsoft's implementation does not suffer from this deficiency. However, if a connection is attempted in which the socket pair is already in the TIME_WAIT state, the connection attempt will fail with error *WSAEADDRINUSE*. One way around this (besides waiting for the socket pair that is using that local port to leave the TIME_WAIT state) is to use the socket option *SO_REUSEADDR*. Chapter 9 covers the *SO_REUSEADDR* option in detail.

The last point of discussion for socket states is the passive closure. In this scenario, an application receives a FIN packet from the peer and responds with an ACK packet. At this point, the application's socket changes to the CLOSE_WAIT state. Because the peer has closed its end, it can't send any more data, but the application still can until it also closes its end of the connection. To close its end of the connection, the application sends its own FIN, causing the application's TCP socket state to become LAST_ACK. After the application receives an ACK packet from the peer, the application's socket reverts to the CLOSED state.

For more information regarding the TCP/IP protocol, consult RFC 793. This RFC and others can be found at http://www.rfc-editor.org.

connect and *WSAConnect*

The only new step is the connect. This is accomplished by calling either *connect* or *WSAConnect*. First we'll look at the Winsock 1 version of this function, which is defined as

```
int connect(
    SOCKET s,
    const struct sockaddr FAR* name,
    int namelen
);
```

The parameters are fairly self-explanatory: *s* is the valid TCP socket on which to establish the connection, *name* is the socket address structure (*SOCKADDR_IN*) for TCP that describes the server to connect to, and *namelen* is the length of the *name* variable. The Winsock 2 version is defined as

```
int WSAConnect(
    SOCKET s,
    const struct sockaddr FAR * name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS
);
```

The first three parameters are exactly the same as the *connect* API function. The next two, *lpCallerData* and *lpCalleeData,* are string buffers used to send and receive data at the time of the connection request. The *lpCallerData* parameter is a pointer to a buffer that holds data the client sends to the server with the connection request. The *lpCalleeData* parameter points to a buffer that will be filled with any data sent back from the server at the time of connection setup. Both of these variables are *WSABUF* structures, so the *len* field needs to be set to the length of data in the *buf* field that is to be transferred in the case of *lpCallerData.* For *lpCalleeData*, the *len* field refers to the length of the buffer in *buf* that can receive data back from the server. The last two parameters, *lpSQOS* and *lpGQOS*, refer to QOS structures that define the bandwidth requirements for both sending and receiving data on the connection to be established. The parameter *lpSQOS* is used to specify requirements for the socket *s*, while *lpGQOS* specifies the requirements for socket groups. Socket groups are not currently supported. A null value for *lpSQOS* indicates no application-specific QOS.

If the computer you're attempting to connect to does not have a process listening on the given port, the *connect* call fails with the error *WSAECONNREFUSED*. The other error you might encounter is *WSAETIMEDOUT*, which occurs if the destination you're trying to reach is unavailable (either because of a communication-hardware failure on the route to the host or because the host is not currently on the network).

Data Transmission

Sending and receiving data is what network programming is all about. For sending data on a connected socket, there are two API functions: *send* and *WSASend*. The second function is specific to Winsock 2. Likewise, two functions are for receiving data on a connected socket: *recv* and *WSARecv*. The latter is also a Winsock 2 call.

An important thing to keep in mind is that all buffers associated with sending and receiving data are of the simple *char* type. That is, there are no UNICODE versions of these functions. This is especially significant on Windows CE, as it uses UNICODE by default. In situations in which you are using UNICODE, you have the option of sending a character string as is or casting it as a *char **. The catch is that if you use the string length function to tell the Winsock API functions how many characters to send or receive, you must multiply this value by 2 because each character occupies 2 bytes of the string array. The other option is to use *WideCharToMultiByte* to convert UNICODE to ASCII before passing the string data to the Winsock API functions.

Additionally, the error code returned by all send and receive functions is *SOCKET_ERROR*. Once an error is returned, call *WSAGetLastError* to obtain extended error information. The most common errors encountered are *WSAECONNABORTED* and *WSAECONNRESET*. Both of these deal with the connection being closed—either through a timeout or through the peer closing the connection. Another common error is *WSAEWOULDBLOCK*, which is normally encountered when either nonblocking or asynchronous sockets are used. This error basically means that the specified function cannot be completed at this time. In Chapter 8, we will describe various Winsock I/O methods that can help you avoid some of these errors.

send and *WSASend*

The first API function to send data on a connected socket is *send*, which is prototyped as

```
int send(
    SOCKET s,
    const char FAR * buf,
    int len,
    int flags
);
```

The *SOCKET* parameter is the connected socket to send the data on. The second parameter, *buf*, is a pointer to the character buffer that contains the data to be sent. The third parameter, *len*, specifies the number of characters in the buffer to send. Finally, the *flags* parameter can be either 0, *MSG_DONTROUTE*, or *MSG_OOB*. Alternatively, the *flags* parameter can be a bitwise ORing of any of those flags. The *MSG_DONTROUTE* flag tells the transport not to route the packets it sends. It is up to the underlying transport to honor this request (for example, if the transport doesn't support this option, it will be ignored). The *MSG_OOB* flag signifies that the data should be sent out of band.

On a good return, *send* returns the number of bytes sent; otherwise, if an error occurs, *SOCKET_ERROR* is returned. A common error is *WSAECONNABORTED,* which occurs when the virtual circuit terminates because of a timeout failure or a protocol error. When this occurs, the socket should be closed, as it is no longer usable. The error *WSAECONNRESET* occurs when the application on the remote host resets the virtual circuit by executing a hard close or terminating unexpectedly, or when the remote host is rebooted. Again, the socket should be closed after this error occurs. The last common error is *WSAETIMEDOUT*, which occurs when the connection is dropped because of a network failure or the remote connected system going down without notice.

The Winsock 2 version of the *send* API function, *WSASend*, is defined as

```
int WSASend(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

The socket is a valid handle to a connection session. The second parameter is a pointer to one or more *WSABUF* structures. This can be either a single structure or an array of such structures. The third parameter indicates the number of *WSABUF* structures being passed. Remember that each *WSABUF* structure is itself a character buffer and the length of that buffer. You might wonder why you would want to send more than one buffer at a time. This is called scatter-gather I/O and will be discussed later in this chapter; however, in the case of data sent using multiple buffers on a connected socket, each buffer is sent from the first to the last *WSABUF* structure in the array. The *lpNumberOfBytesSent* is a pointer to a *DWORD* that on return from the *WSASend* call contains the total number of bytes sent. The *dwFlags* parameter is equivalent to its counterpart in *send*. The last two parameters, *lpOverlapped* and *lpCompletionROUTINE*, are used for overlapped I/O. Overlapped I/O is one of the asynchronous I/O models supported by Winsock and is discussed in detail in Chapter 8.

The *WSASend* function sets *lpNumberOfBytesSent* to the number of bytes written. The function returns 0 on success and *SOCKET_ERROR* on any error, and generally encounters the same errors as the *send* function.

WSASendDisconnect

This function is rather specialized and not generally used. The function prototype is

```
int WSASendDisconnect (
    SOCKET s,
    LPWSABUF lpOUT boundDisconnectData
);
```

> Out-of-Band Data
>
> When an application on a connected stream socket needs to send data that is more important than regular data on the stream, it can mark the important data as out-of-band (OOB) data. The application on the other end of a connection can receive and process OOB data through a separate logical channel that is conceptually independent of the data stream.
>
> In TCP, OOB data is implemented via an urgent 1-bit marker (called URG) and a 16-bit pointer in the TCP segment header that identify a specific downstream byte as urgent data. Two specific ways of implementing urgent data currently exist for TCP. RFC 793, which describes TCP and introduces the concept of urgent data, indicates that the urgent pointer in the TCP header is a positive offset to the byte that follows the urgent data byte. However, RFC 1122 describes the urgent offset as pointing to the urgent byte itself.
>
> The Winsock specification uses the term OOB to refer to both protocol-independent OOB data and TCP's implementation of OOB data (urgent data). In order to check whether pending data contains urgent data, you must call the *ioctlsocket* function with the *SIOCATMARK* option. Chapter 9 discusses how to use *SIOCATMARK*.
>
> Winsock provides several methods for obtaining the urgent data. Either the urgent data is inlined so that it appears in the normal data stream, or in-lining can be turned off so that a discrete call to a receive function returns only the urgent data. The socket option *SO_OOBINLINE*, also discussed in detail in Chapter 9, controls the behavior of OOB data.
>
> Telnet and Rlogin use urgent data for several reasons. However, unless you plan on writing your own Telnet or Rlogin, you should stay away from urgent data. It's not well defined and might be implemented differently on platforms other than Win32. If you require a method of signaling the peer for urgent reasons, implement a separate control socket for this urgent data and reserve the main socket connection for normal data transfers.

The function initiates a shutdown of the socket and sends disconnect data. Of course, this function is available only to those transport protocols that support graceful close and disconnect data. None of the transport providers currently support disconnect data. The *WSASendDisconnect* function behaves like a call to the *shutdown* function with an *SD_SEND* argument, but it also sends the data contained in its *boundDisconnectData* parameter. Subsequent sends are not allowed on the socket. Upon failure, *WSASendDisconnect* returns *SOCKET_ERROR*. This function can encounter some of the same errors as the *send* function.

recv and *WSARecv*

The *recv* function is the most basic way to accept incoming data on a connected socket. This function is defined as

```
int recv(
    SOCKET s,
    char FAR* buf,
    int len,
    int flags
);
```

The first parameter, *s*, is the socket on which data will be received. The second parameter, *buf*, is the character buffer that will receive the data, while *len* is either the number of bytes you want to receive or the size of the buffer, *buf*. Finally, the *flags* parameter can be one of the following values: 0, *MSG_PEEK*, or *MSG_OOB*. Additionally, you can bitwise OR any one of these flags together. Of course, 0 specifies no special actions. *MSG_PEEK* causes the data that is available to be copied into the supplied receive buffer, but this data is not removed from the system's buffer. The number of bytes pending is also returned.

Message peeking is bad. Not only does it degrade performance, as you now need to make two system calls (one to peek and one without the *MSG_PEEK* flag to actually remove the data), but it is also unreliable under certain circumstances. The data returned might not reflect the entire amount available. Also, by leaving data in the system buffers, the system has less and less space to contain incoming data. As a result, the system reduces the TCP window size for all senders. This prevents your application from achieving the maximum possible throughput. The best thing to do is to copy all the data you can into your own buffer and manipulate it there. You have seen the *MSG_OOB* flag before in the discussion on sending data. Refer to the previous section for more information.

There are some considerations when using *recv* on a message- or datagram-based socket. In the event that the data pending is larger than the supplied buffer, the buffer is filled with as much data as it will contain. In this event, the *recv* call generates the error *WSAEMSGSIZE.* Note that the message-size error occurs with message-oriented protocols. Stream protocols buffer incoming data and will return as much data as the application requests, even if the amount of pending data is greater. Thus, for streaming protocols you will not encounter the *WSAEMSGSIZE* error.

The *WSARecv* function adds some new capabilities over *recv,* such as overlapped I/O and partial datagram notifications. The definition of *WSARecv* is

```
int WSARecv(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

Parameter *s* is the connected socket. The second and third parameters are the buffers to receive the data. The *lpBuffers* parameter is an array of *WSABUF* structures, while *dwBufferCount* indicates the number of *WSABUF* structures in the array. The *lpNumberOfBytesReceived* parameter points to the number of bytes received by this call if the receive operation completes immediately. The *lpFlags* parameter can be one of the values *MSG_PEEK*, *MSG_OOB*, or *MSG_PARTIAL* or a bitwise ORed combination of those values. The *MSG_PARTIAL* flag has several different meanings depending on where it is used or encountered. For message-oriented protocols, this flag is set upon return from *WSARecv* (if the entire message could not be returned in this call because of insufficient buffer space). In this case, subsequent *WSARecv* calls set this flag until the entire message is returned, when the *MSG_PARTIAL* flag is cleared. If this flag is passed as an input parameter, the receive operation should complete as soon as data is available, even if it is only a portion of the entire message. The *MSG_PARTIAL* flag is used only with message-oriented protocols, not with streaming ones. Additionally, not all protocols support partial messages. The protocol entry for each protocol contains a flag indicating whether it supports this feature. See Chapter 5 for more information. The *lpOverlapped* and *lpCompletionROUTINE* parameters are used in overlapped I/O operations, discussed in Chapter 8.

WSARecvDisconnect

This function is the opposite of *WSASendDisconnect* and is defined as follows:

```
int WSARecvDisconnect(
    SOCKET s,
    LPWSABUF lpInboundDisconnectData
);
```

Like its sending counterpart, the parameters are the connected socket handle and a valid *WSABUF* structure with the data to be received. The data received can only be disconnect data sent by a *WSASendDisconnect* on the other side; it cannot be used to receive normal data. Additionally, once the data is received, this function disables reception from the remote party, which is equivalent to calling the *shutdown* function with *SD_RECV*.

WSARecvEx

The *WSARecvEx* function is a Microsoft-specific extension of Winsock 1 and is identical to the *recv* function except that the *flags* parameter is passed by reference. This allows the underlying provider to set the *MSG_PARTIAL* flag. The function prototype is as follows:

```
int PASCAL FAR WSARecvEx(
    SOCKET s,
    char FAR * buf,
    int len,
    int *flags
);
```

The *MSG_PARTIAL* flag is returned in the *flags* parameter if the data received is not a complete message. This flag is of interest for message-oriented (nonstream) protocols. If the *MSG_PARTIAL* flag is passed as a part of the *flags* parameter and a partial message is received, the call returns immediately with that data. If the supplied receive buffer is not large enough to hold an entire message, *WSARecvEx* fails with the *WSAEMSGSIZE* error and the remaining data is truncated. Note that the difference between a *MSG_PARTIAL* flag and a *WSAEMSGSIZE* error is that with the error, the whole message arrives but the supplied data buffer is too small to receive it. The *MSG_PEEK* and *MSG_OOB* flags can also be used with *WSARecvEx*.

Stream Protocols

Because most connection-oriented protocols are also streaming protocols, we'll mention stream protocols here. The main thing to be aware of with *any* function that sends or receives data on a stream socket is that you are not guaranteed to read or write the amount of data you request. Let's say you have a character buffer with 2048 bytes of data you want to send with the *send* function. The code to send this is

```
char sendbuff[2048];
int  nBytes = 2048;

// Fill sendbuff with 2048 bytes of data

// Assume s is a valid, connected stream socket
ret = send(s, sendbuff, nBytes, 0);
```

It is possible for *send* to return having sent less than 2048 bytes. The *ret* variable will be set to the number of bytes sent because the system allocates a certain amount of buffer space for each socket to send and receive

data. In the case of sending data, the internal buffers hold data to be sent until such time as the data can be placed on the wire. Several common situations can cause this. For example, simply transmitting a huge amount of data will cause these buffers to become filled quickly. Also, for TCP/IP, there is what is known as the window size. The receiving end will adjust this window size to indicate how much data it can receive. If the receiver is being flooded with data, it might set the window size to 0 in order to catch up with the pending data. This will force the sender to stop until it receives a new window size greater than 0. In the case of our *send* call, there might only be buffer space to hold 1024 bytes, in which case you would have to resubmit the remaining 1024 bytes. The following code ensures that all your bytes are sent.

```
char sendbuff[2048];
int  nBytes = 2048,
     nLeft,
     idx;

// Fill sendbuff with 2048 bytes of data

// Assume s is a valid, connected stream socket
nLeft = nBytes;
idx = 0;
while (nLeft > 0)
{
    ret = send(s, &sendbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Error
    }
    nLeft -= ret;
    idx += ret;
}
```

The foregoing holds true for receiving data on a stream socket but is less significant. Because stream sockets are a continuous stream of data, when an application reads it isn't generally concerned with how much data it should read. If your application requires discrete messages over a stream protocol, you might have to do a little work. If all the messages are the same size, life is pretty simple, and the code for reading, say, 512-byte messages would look like this:

```
char    recvbuff[1024];
int     ret,
        nLeft,
        idx;

nLeft = 512;
idx = 0;
while (nLeft > 0)
{
    ret = recv(s, &recvbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Error
    }
    idx += ret;
    nLeft -= ret;
```

```
}
```

Things get a little complicated if your message sizes vary. It is necessary to impose your own protocol to let the receiver know how big the forthcoming message will be. For example, the first 4 bytes written to the receiver will always be the integer size in bytes of the forthcoming message. The receiver will start every read by looking at the first 4 bytes, converting them to an integer, and determining how many additional bytes that message comprises.

> Scatter-Gather I/O
>
> Scatter-gather support is a concept originally introduced in Berkeley Sockets with the functions *recv* and *writev*. This feature is available with the Winsock 2 functions *WSARecv*, *WSARecvFrom*, *WSASend*, and *WSASendTo*. It is most useful for applications that send and receive data that is formatted in a very specific way. For example, messages from a client to a server might always be composed of a fixed 32-byte header specifying some operation, followed by a 64-byte data block and terminated with a 16-byte trailer. In this example, *WSASend* can be called with an array of three *WSABUF* structures, each corresponding to the three message types. On the receiving end, *WSARecv* is called with three *WSABUF* structures, each containing data buffers of 32 bytes, 64 bytes, and 16 bytes.
>
> When using stream-based sockets, scatter-gather operations simply treat the supplied data buffers in the *WSABUF* structures as one contiguous buffer. Also, the receive call might return before all buffers are full. On message-based sockets, each call to a receive operation receives a single message up to the buffer size supplied. If the buffer space is insufficient, the call fails with *WSAEMSGSIZE* and the data is truncated to fit the available space. Of course, with protocols that support partial messages, the *MSG_PARTIAL* flag can be used to prevent data loss.

Breaking the Connection

Once you are finished with a socket connection, you must close the connection and release any resources associated with that socket handle. To actually release the resources associated with an open socket handle, use the *closesocket* call. Be aware, however, that *closesocket* can have some adverse affects—depending on how it is called—that can lead to data loss. For this reason, a connection should be gracefully terminated with the *shutdown* function before a call to the *closesocket* function. These two API functions are discussed next.

shutdown

To ensure that all data an application sends is received by the peer, a well-written application should notify the receiver that no more data is to be sent. Likewise, the peer should do the same. This is known as a graceful close and is performed by the *shutdown* function, defined as

```
int shutdown(
    SOCKET s,
    int how
);
```

The *how* parameter can be *SD_RECEIVE*, *SD_SEND*, or *SD_BOTH*. For *SD_RECEIVE*, subsequent calls to any receive function on the socket are disallowed. This has no effect on the lower protocol layers. Additionally for TCP sockets, if data is queued for receive or if data subsequently arrives, the connection is reset. However, on UDP sockets incoming data is still accepted and queued. For *SD_SEND*, subsequent calls to any send function are disallowed. For TCP sockets, this causes a FIN packet to be generated after all data is sent and acknowledged by the receiver. Finally, specifying *SD_BOTH* disables both sends and receives.

closesocket

The *closesocket* function closes a socket and is defined as

```
int closesocket (SOCKET s);
```

Calling *closesocket* releases the socket descriptor and any further calls using the socket fail with *WSAENOTSOCK.* If there are no other references to this socket, all resources associated with the descriptor are released. This includes discarding any queued data.

Pending asynchronous calls issued by any thread in this process are canceled without posting any notification messages. Pending overlapped operations are also canceled. Any event, completion routine, or completion port that is associated with the overlapped operation is performed but will fail with the error *WSA_OPERATION_ABORTED*. Asynchronous and nonblocking I/O models are discussed in greater depth in Chapter 8. Additionally, one other factor influences the behavior of *closesocket*: whether the socket option *SO_LINGER* has been set. Consult the description for the *SO_LINGER* option in Chapter 9 for a complete explanation.

Putting It All Together

You might be a bit overwhelmed by the multitude of functions for sending and receiving data, but in reality most applications only need either *recv* or *WSARecv* for receiving data and either *send* or *WSASend* for sending. The other functions are specialized with unique features not commonly used (or supported by the transport protocols). With this said, we'll discuss a simple client/server example using the principles and functions we've covered so far. Figure 7-3 contains the code for a simple echo server. This application creates a socket, binds to a local IP interface and port, and listens for client connections. Upon receipt of a client connection request, a new socket is created that is passed into a client thread that is spawned. The thread simply reads data and sends it back to the client.

Figure 7-3. *Echo server code*

```
// Module Name: Server.c
//
// Description:
//    This example illustrates a simple TCP server that accepts
//    incoming client connections. Once a client connection is
//    established, a thread is spawned to read data from the
//    client and echo it back (if the echo option is not
//    disabled).
//
// Compile:
//    cl -o Server Server.c ws2_32.lib
//
// Command line options:
//    server [-p:x] [-i:IP] [-o]
//           -p:x      Port number to listen on
//           -i:str    Interface to listen on
//           -o        Receive only; don't echo the data back
//
#include <winsock2.h>

#include <stdio.h>
#include <stdlib.h>
```

```
#define DEFAULT_PORT        5150
#define DEFAULT_BUFFER      4096

int    iPort      = DEFAULT_PORT; // Port to listen for clients on

BOOL   bInterface = FALSE,   // Listen on the specified interface
       bRecvOnly  = FALSE;   // Receive data only; don't echo back
char   szAddress[128];       // Interface to listen for clients on

//
// Function: usage
//
// Description:
//    Print usage information and exit
//
void usage()
{
    printf("usage: server [-p:x] [-i:IP] [-o]\n\n");
    printf("       -p:x      Port number to listen on\n");
    printf("       -i:str    Interface to listen on\n");
    printf("       -o        Don't echo the data back\n\n");
    ExitProcess(1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some global flags
//    to indicate what actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int i;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'p':
                    iPort = atoi(&argv[i][3]);
                    break;
                case 'i':
                    bInterface = TRUE;
                    if (strlen(argv[i]) > 3)
                        strcpy(szAddress, &argv[i][3]);
                    break;
                case 'o':
                    bRecvOnly = TRUE;
                    break;
                default:
```

```
                    usage();
                    break;
            }
        }
    }
}

//
// Function: ClientThread
//
// Description:
//    This function is called as a thread, and it handles a given
//    client connection.  The parameter passed in is the socket
//    handle returned from an accept() call.  This function reads
//    data from the client and writes it back.
//
DWORD WINAPI ClientThread(LPVOID lpParam)
{
    SOCKET        sock=(SOCKET)lpParam;
    char          szBuff[DEFAULT_BUFFER];
    int           ret,
                  nLeft,
                  idx;

    while(1)
    {
        // Perform a blocking recv() call
        //
        ret = recv(sock, szBuff, DEFAULT_BUFFER, 0);
        if (ret == 0)        // Graceful close
            break;
        else if (ret == SOCKET_ERROR)
        {
            printf("recv() failed: %d\n", WSAGetLastError());
            break;
        }
        szBuff[ret] = '\0';
        printf("RECV: '%s'\n", szBuff);
        //
        // If we selected to echo the data back, do it
        //
        if (!bRecvOnly)
        {
            nLeft = ret;
            idx = 0;
            //
            // Make sure we write all the data
            //
            while(nLeft > 0)
            {
                ret = send(sock, &szBuff[idx], nLeft, 0);
                if (ret == 0)
                    break;
```

```
                else if (ret == SOCKET_ERROR)
                {
                    printf("send() failed: %d\n",
                        WSAGetLastError());
                    break;
                }
                nLeft -= ret;
                idx += ret;
            }
        }
    }
    return 0;
}

//
// Function: main
//
// Description:
//    Main thread of execution. Initialize Winsock, parse the
//    command line arguments, create the listening socket, bind
//    to the local address, and wait for client connections.
//
int main(int argc, char **argv)
{
    WSADATA       wsd;
    SOCKET        sListen,
                  sClient;
    int           iAddrSize;
    HANDLE        hThread;
    DWORD         dwThreadId;
    struct sockaddr_in local,
                       client;

    ValidateArgs(argc, argv);
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("Failed to load Winsock!\n");
        return 1;
    }

    // Create our listening socket
    //
    sListen = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    if (sListen == SOCKET_ERROR)
    {
        printf("socket() failed: %d\n", WSAGetLastError());
        return 1;
    }
    // Select the local interface, and bind to it
    //
    if (bInterface)
    {
        local.sin_addr.s_addr = inet_addr(szAddress);
```

```
        if (local.sin_addr.s_addr == INADDR_NONE)
            usage();
    }
    else
        local.sin_addr.s_addr = htonl(INADDR_ANY);
    local.sin_family = AF_INET;
    local.sin_port = htons(iPort);

    if (bind(sListen, (struct sockaddr *)&local,
            sizeof(local)) == SOCKET_ERROR)
    {
        printf("bind() failed: %d\n", WSAGetLastError());
        return 1;
    }
    listen(sListen, 8);
    //
    // In a continuous loop, wait for incoming clients. Once one
    // is detected, create a thread and pass the handle off to it.
    //
    while (1)
    {
        iAddrSize = sizeof(client);
        sClient = accept(sListen, (struct sockaddr *)&client,
                        &iAddrSize);
        if (sClient == INVALID_SOCKET)
        {
            printf("accept() failed: %d\n", WSAGetLastError());
            break;
        }
        printf("Accepted client: %s:%d\n",
            inet_ntoa(client.sin_addr), ntohs(client.sin_port));

        hThread = CreateThread(NULL, 0, ClientThread,
                    (LPVOID)sClient, 0, &dwThreadId);
        if (hThread == NULL)
        {
            printf("CreateThread() failed: %d\n", GetLastError());
            break;
        }
        CloseHandle(hThread);
    }
    closesocket(sListen);

    WSACleanup();
    return 0;
}
```

The client for this example, provided in Figure 7-4, is even more basic. The client creates a socket, resolves the server name passed into the application, and connects to the server. Once the connection is made, a number of messages are sent. After each send, the client waits for an echo response from the server. The client prints all data read from the socket.

The echo client and server don't fully illustrate the streaming nature of TCP. This is because a read operation

follows every write operation, at least in the client's case. Of course, it is the other way around for the server. Thus, each call to the read function by the server will almost always return the full message that the client sent. Don't be misled by this. If the client's messages become large enough to exceed the maximum transmission unit for TCP, the message will be broken up into separate packets on the wire, in which case the receiver needs to perform a receive call multiple times. In order to better illustrate streaming, run the client and the server with the -o option. This causes the client to only send data and the receiver to only read data. Execute the server like this:

```
server -p:5150 -o
```

and the client like this:

```
client -p:5150 -s:IP -n:10 -o
```

What you'll most likely see is that the client calls *send* 10 times, but the server reads all 10 messages in one or two *recv* calls.

Figure 7-4. *Echo client code*

```
// Module Name: Client.c
//
// Description:
//    This sample is the echo client. It connects to the TCP server,
//    sends data, and reads data back from the server.
//
// Compile:
//    cl -o Client Client.c ws2_32.lib
//
// Command Line Options:
//    client [-p:x] [-s:IP] [-n:x] [-o]
//           -p:x      Remote port to send to
//           -s:IP     Server's IP address or host name
//           -n:x      Number of times to send message
//           -o        Send messages only; don't receive
//
#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_COUNT       20
#define DEFAULT_PORT        5150
#define DEFAULT_BUFFER      2048
#define DEFAULT_MESSAGE     "This is a test of the emergency \
broadcasting system"

char  szServer[128],          // Server to connect to
      szMessage[1024];        // Message to send to sever
int   iPort     = DEFAULT_PORT;  // Port on server to connect to
DWORD dwCount   = DEFAULT_COUNT; // Number of times to send message
BOOL  bSendOnly = FALSE;         // Send data only; don't receive
```

```
//
// Function: usage:
//
// Description:
//    Print usage information and exit
//
void usage()
{
    printf("usage: client [-p:x] [-s:IP] [-n:x] [-o]\n\n");
    printf("       -p:x      Remote port to send to\n");
    printf("       -s:IP     Server's IP address or host name\n");
    printf("       -n:x      Number of times to send message\n");
    printf("       -o        Send messages only; don't receive\n");
    ExitProcess(1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some global flags
//    to indicate what actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int               i;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'p':        // Remote port
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 's':       // Server
                    if (strlen(argv[i]) > 3)
                        strcpy(szServer, &argv[i][3]);
                    break;
                case 'n':       // Number of times to send message
                    if (strlen(argv[i]) > 3)
                        dwCount = atol(&argv[i][3]);
                    break;
                case 'o':       // Only send message; don't receive
                    bSendOnly = TRUE;
                    break;
                default:
                    usage();
                    break;
            }
```

```
        }
    }
}

//
// Function: main
//
// Description:
//    Main thread of execution. Initialize Winsock, parse the
//    command line arguments, create a socket, connect to the
//    server, and then send and receive data
//
int main(int argc, char **argv)
{
    WSADATA       wsd;
    SOCKET        sClient;
    char          szBuffer[DEFAULT_BUFFER];
    int           ret,
                  i;

    struct sockaddr_in server;
    struct hostent    *host = NULL;

    // Parse the command line, and load Winsock
    //
    ValidateArgs(argc, argv);
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("Failed to load Winsock library!\n");
        return 1;
    }
    strcpy(szMessage, DEFAULT_MESSAGE);
    //
    // Create the socket, and attempt to connect to the server
    //
    sClient = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sClient == INVALID_SOCKET)
    {
        printf("socket() failed: %d\n", WSAGetLastError());
        return 1;
    }
    server.sin_family = AF_INET;
    server.sin_port = htons(iPort);
    server.sin_addr.s_addr = inet_addr(szServer);
    //
    // If the supplied server address wasn't in the form
    // "aaa.bbb.ccc.ddd," it's a host name, so try to resolve it
    //
    if (server.sin_addr.s_addr == INADDR_NONE)
    {
        host = gethostbyname(szServer);
        if (host == NULL)
        {
```

```
            printf("Unable to resolve server: %s\n", szServer);
            return 1;
        }
        CopyMemory(&server.sin_addr, host->h_addr_list[0],
            host->h_length);
    }
    if (connect(sClient, (struct sockaddr *)&server,
        sizeof(server)) == SOCKET_ERROR)
    {
        printf("connect() failed: %d\n", WSAGetLastError());
        return 1;
    }
    // Send and receive data
    //
    for(i = 0; i < dwCount; i++)
    {
        ret = send(sClient, szMessage, strlen(szMessage), 0);
        if (ret == 0)
            break;
        else if (ret == SOCKET_ERROR)
        {
            printf("send() failed: %d\n", WSAGetLastError());
            break;
        }
        printf("Send %d bytes\n", ret);
        if (!bSendOnly)
        {
            ret = recv(sClient, szBuffer, DEFAULT_BUFFER, 0);
            if (ret == 0)        // Graceful close
                break;
            else if (ret == SOCKET_ERROR)
            {
                printf("recv() failed: %d\n", WSAGetLastError());
                break;
            }
            szBuffer[ret] = '\0';
            printf("RECV [%d bytes]: '%s'\n", ret, szBuffer);
        }
    }
    closesocket(sClient);

    WSACleanup();
    return 0;
}
```

Connectionless Protocols

Connectionless protocols behave differently than connection-oriented protocols, so the method for sending and receiving data is substantially different. First we'll discuss the receiver (or server, if you prefer) because the connectionless receiver requires little change when compared with the session-oriented servers. Following that we'll look at the sender.

Receiver

For a process to receive data on a connectionless socket, the steps are simple. First create the socket with either *socket* or *WSASocket*. Next bind the socket to the interface on which you wish to receive data. This is done with the *bind* function (exactly like the session-oriented example). The difference with connectionless sockets is that you do not call *listen* or *accept*. Instead, you simply wait to receive the incoming data. Because there is no connection, the receiving socket can receive datagrams originating from any machine on the network. The simplest of the receive functions is *recvfrom*, which is defined as

```
int recvfrom(
    SOCKET s,
    char FAR* buf,
    int len,
    int flags,
    struct sockaddr FAR* from,
    int FAR* fromlen
);
```

The first four parameters are the same as *recv*, including the possible values for *flags—MSG_OOB* and *MSG_PEEK*. The same warnings for using the *MSG_PEEK* flag also apply to connectionless sockets. The *from* parameter is a *SOCKADDR* structure for the given protocol of the listening socket, with *fromlen* pointing to the size of the address structure. When the API call returns with data, the *SOCKADDR* structure is filled with the address of the workstation that sent the data.

The Winsock 2 version of the *recvfrom* function is *WSARecvFrom*. The prototype for this function is

```
int WSARecvFrom(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    struct sockaddr FAR * lpFrom,
    LPINT lpFromlen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

The difference is the use of *WSABUF* structures for receiving the data. You can supply one or more *WSABUF* buffers to *WSARecvFrom* with *dwBufferCount* indicating this. By supplying multiple buffers, scatter-gather I/O is possible. The total number of bytes read is returned in *lpNumberOfBytesRecvd*. When you call *WSARecvFrom*, the *lpFlags* parameter can be 0 for no options, *MSG_OOB*, *MSG_PEEK*, or *MSG_PARTIAL*. These flags can be ORed together. If *MSG_PARTIAL* is specified when the function is called, the provider knows to return data even if only a

partial message has been received. Upon return, the flag *MSG_PARTIAL* is set if only a partial message was received. Upon return, *WSARecvFrom* will set the *lpFrom* parameter (a pointer to a *SOCKADDR* structure) to the address of the sending machine. Again, *lpFromLen* points to the size of the *SOCKADDR* structure, except that in this function it is a pointer to a *DWORD*. The last two parameters, *lpOverlapped* and *lpCompletionROUTINE*, are used for overlapped I/O (which we'll discuss in the next chapter).

Another method of receiving (and sending) data on a connectionless socket is to establish a connection. This might sound strange, but it's not quite what it sounds like. Once a connectionless socket is created, you can call *connect* or *WSAConnect* with the *SOCKADDR* parameter set to the address of the remote machine to communicate with. No actual connection is made, however. The socket address passed into a connect function is associated with the socket so that *recv* and *WSARecv* can be used instead of *recvfrom* or *WSARecvFrom* because the data's origin is known. The ability to connect a datagram socket is handy if you intend to communicate with only one endpoint at a time in your application.

Sender

To send data on a connectionless socket, there are two options. The first, and simplest, is to create a socket and call either *sendto* or *WSASendTo*. We'll cover *sendto* first, which is defined as

```
int sendto(
    SOCKET s,
    const char FAR * buf,
    int len,
    int flags,
    const struct sockaddr FAR * to,
    int tolen
);
```

The parameters are the same as *recvfrom* except that *buf* is the buffer of data to send and *len* indicates how many bytes to send. Also, the *to* parameter is a pointer to a *SOCKADDR* structure with the destination address of the workstation to receive the data. The Winsock 2 function *WSASendTo* can also be used. This function is defined as

```
int WSASendTo(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    const struct sockaddr FAR * lpTo,
    int iToLen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

Again, *WSASendTo* is similar to its ancestor. This function takes a pointer to one or more *WSABUF* structures with data to send to the recipient as the *lpBuffers* parameter, with *dwBufferCount* indicating how many structures are present. You can send multiple *WSABUF* structures to enable scatter-gather I/O. Before returning, *WSASendTo* sets the fourth parameter, *lpNumberOfBytesSent*, to the number of bytes actually sent to the receiver. The *lpTo* parameter is a *SOCKADDR* structure for the given protocol, with the recipient's address. The *iToLen* parameter is the length of the *SOCKADDR* structure. The last two parameters, *lpOverlapped* and *lpCompletionROUTINE*, are used for overlapped I/O (discussed in Chapter 8).

As with receiving data, a connectionless socket can be connected to an endpoint address and data can be sent

with *send* and *WSASend*. Once this association is established, you cannot go back to using *sendto* or *WSASendTo* with an address other than the address passed to one of the connect functions. If you do attempt to send data to a different address, the call will fail with *WSAEISCONN*. The only way to disassociate the socket handle from that destination is to call *closesocket* on the handle and create a new socket.

Message-Based Protocols

Just as most connection-oriented protocols are also streaming, connectionless protocols are almost always message-based. Thus, there are some considerations when you're sending and receiving data. First, because message-based protocols preserve data boundaries, data submitted to a send function blocks until completed. For asynchronous or nonblocking I/O modes, if a send cannot be completely satisfied, the send function returns with the error *WSAEWOULDBLOCK*. This means that the underlying system was not able to process that data and you should attempt the send call again at a later time. This scenario will be discussed in greater detail in the next chapter. The main thing to remember is that with message-based protocols, the write can occur only as an autonomous action.

On the flip side, a call to a receive function must supply a sufficiently large buffer. If the supplied buffer is not large enough, the receive call fails with the error *WSAEMSGSIZE*. If this occurs, the buffer is filled to its capacity, but the remaining data is discarded. The truncated data cannot be retrieved. The only exception is for protocols that do support partial messages, such as the AppleTalk PAP protocol. Prior to returning, the *WSARecvEx* function sets its in-out *flag* parameter to *MSG_PARTIAL* when it receives only part of a message.

For datagrams based on protocols supporting partial messages, consider using one of the *WSARecv* functions. When you make a call to *recv*, there is no notification that the data read is only a partial message. It is up to the programmer to implement a method for the receiver to determine whether the entire message has been read. Subsequent calls to *recv* return other pieces of the datagram. Because of this limitation, it can be convenient to use the *WSARecvEx* function, which allows the setting and reading of the *MSG_PARTIAL* flag to indicate whether the entire message was read. The Winsock 2 functions *WSARecv* and *WSARecvFrom* also support this flag. See the descriptions for *WSARecv*, *WSARecvEx*, and *WSARecvFrom* for additional information about this flag.

Finally, let's take a look at one of the more frequently asked questions concerning sending UDP/IP messages on machines with multiple network interfaces: what happens when a UDP socket is bound explicitly to a local IP interface and datagrams are sent? With UDP sockets, you don't really bind to the network interface; you create an association whereby the IP interface that is bound becomes the source IP address of UDP datagrams sent. The routing table actually determines which physical interface the datagram is transmitted on. If you do not call *bind* but instead either use *sendto*/*WSASendTo* or perform a connect first, the network stack automatically picks the best local IP address based on the routing table. What this means is that if you explicitly bind first, the source IP address could be incorrect. That is, the source IP might not be the IP address of the interface on which the datagram was actually sent.

Releasing Socket Resources

Because there is no connection with connectionless protocols, there is no formal shutdown or graceful closing of the connection. When the sender or the receiver is finished sending or receiving data, it simply calls the *closesocket* function on the socket handle. This releases any associated resources allocated to the socket.

Putting It All Together

Now that we have covered the necessary steps for sending and receiving data on a connectionless socket, let's look at some actual code that performs this procedure. Figure 7-5 shows the first example, a connectionless receiver. The code illustrates how to receive a datagram on either the default interface or a specified local interface.

Figure 7-5. *Connectionless receiver*

```
// Module Name: Receiver.c
//
// Description:
//    This sample receives UDP datagrams by binding to the specified
```

```
//    interface and port number and then blocking on a recvfrom()
//    call
//
// Compile:
//    cl -o Receiver Receiver.c ws2_32.lib
//
// Command Line Options:
//    sender [-p:int] [-i:IP][-n:x] [-b:x]
//           -p:int   Local port
//           -i:IP    Local IP address to listen on
//           -n:x     Number of times to send message
//           -b:x     Size of buffer to send
//
#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_PORT            5150
#define DEFAULT_COUNT           25
#define DEFAULT_BUFFER_LENGTH   4096

int   iPort    = DEFAULT_PORT;          // Port to receive on
DWORD dwCount  = DEFAULT_COUNT,         // Number of messages to read
      dwLength = DEFAULT_BUFFER_LENGTH; // Length of receiving buffer
BOOL  bInterface = FALSE;               // Use an interface other than
                                        // default
char  szInterface[32];            // Interface to read datagrams from

//
// Function: usage:
//
// Description:
//    Print usage information and exit
//
void usage()
{
    printf("usage: sender [-p:int] [-i:IP][-n:x] [-b:x]\n\n");
    printf("       -p:int   Local port\n");
    printf("       -i:IP    Local IP address to listen on\n");
    printf("       -n:x     Number of times to send message\n");
    printf("       -b:x     Size of buffer to send\n\n");
    ExitProcess(1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some global flags to
//    indicate what actions to perform
//
void ValidateArgs(int argc, char **argv)
{
```

```
    int               i;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'p':   // Local port
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 'n':   // Number of times to receive message
                    if (strlen(argv[i]) > 3)
                        dwCount = atol(&argv[i][3]);
                    break;
                case 'b':   // Buffer size
                    if (strlen(argv[i]) > 3)
                        dwLength = atol(&argv[i][3]);
                    break;
                case 'i':    // Interface to receive datagrams on
                    if (strlen(argv[i]) > 3)
                    {
                        bInterface = TRUE;
                    strcpy(szInterface, &argv[i][3]);
                    }
                    break;
                default:
                    usage();
                    break;
            }
        }
    }
}

//
// Function: main
//
// Description:
//    Main thread of execution. Initialize Winsock, parse the command
//    line arguments, create a socket, bind it to a local interface
//    and port, and then read datagrams.
//
int main(int argc, char **argv)
{
    WSADATA        wsd;
    SOCKET         s;
    char          *recvbuf = NULL;
    int            ret,
                   i;
    DWORD          dwSenderSize;
    SOCKADDR_IN    sender,
                   local;
```

```
    // Parse arguments and load Winsock
    //
    ValidateArgs(argc, argv);

    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("WSAStartup failed!\n");
        return 1;
    }
    // Create the socket, and bind it to a local interface and port
    //
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s == INVALID_SOCKET)
    {
        printf("socket() failed; %d\n", WSAGetLastError());
        return 1;
    }
    local.sin_family = AF_INET;
    local.sin_port = htons((short)iPort);
    if (bInterface)
        local.sin_addr.s_addr = inet_addr(szInterface);
    else
        local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (SOCKADDR *)&local, sizeof(local)) == SOCKET_ERROR)
    {
        printf("bind() failed: %d\n", WSAGetLastError());
        return 1;
    }
    // Allocate the receive buffer
    //
    recvbuf = GlobalAlloc(GMEM_FIXED, dwLength);
    if (!recvbuf)
    {
        printf("GlobalAlloc() failed: %d\n", GetLastError());
        return 1;
    }
    // Read the datagrams
    //
    for(i = 0; i < dwCount; i++)
    {
        dwSenderSize = sizeof(sender);
        ret = recvfrom(s, recvbuf, dwLength, 0,
                (SOCKADDR *)&sender, &dwSenderSize);
        if (ret == SOCKET_ERROR)
        {
            printf("recvfrom() failed; %d\n", WSAGetLastError());
            break;
        }
        else if (ret == 0)
            break;
        else
        {
```

```
            recvbuf[ret] = '\0';
            printf("[%s] sent me: '%s'\n",
                inet_ntoa(sender.sin_addr), recvbuf);
        }
    }
    closesocket(s);

    GlobalFree(recvbuf);
    WSACleanup();
    return 0;
}
```

Receiving a datagram is easy. First you create a socket, and then you bind the socket to the local interface. If you bind to the default interface, you can find its IP address by using the *getsockname* function. This function simply returns the *SOCKADDR_IN* structure associated with the given socket passed to it, indicating the interface on which the socket is bound. After that, it's just a matter of making calls to *recvfrom* in order to read the incoming data. Note that we're using *recvfrom* because we are not concerned with partial messages; the UDP protocol does not support partial messages. In fact, when the TCP/IP stack receives pieces of a larger datagram message, it waits until all parts have been assembled. If the pieces are out of order or one or more pieces are missing, the stack discards the whole message.

Figure 7-6 provides the code for the next example, a connectionless sender. The sender example has quite a few more options than the receiver does. The necessary parameters are the IP address and port of the remote recipient. The -c option specifies whether to make a call to *connect* first; the default behavior is not to make that call. Again, the steps are simple. First create the socket. If the -c option is present, make a call to *connect* with the remote recipient's address and port number. This is followed by calls to *send*. If no connect is performed, simply start sending data to the recipient after socket creation with the *sendto* function.

Figure 7-6. *Connectionless sender*

```
// Module Name: Sender.c
//
// Description:
//    This sample sends UDP datagrams to the specified recipient.
//    The -c option first calls connect() to associate the
//    recipient's IP address with the socket handle so that the
//    send() function can be used as opposed to the sendto() call.
//
// Compile:
//    cl -o Sender Sender.c ws2_32.lib
//
// Command line options:
//    sender [-p:int] [-r:IP] [-c] [-n:x] [-b:x] [-d:c]
//           -p:int   Remote port
//           -r:IP    Recipient's IP address or host name
//           -c       Connect to remote IP first
//           -n:x     Number of times to send message
//           -b:x     Size of buffer to send
//           -d:c     Character to fill buffer with
//
#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#define DEFAULT_PORT            5150
#define DEFAULT_COUNT           25
#define DEFAULT_CHAR            'a'
#define DEFAULT_BUFFER_LENGTH   64

BOOL  bConnect = FALSE;                 // Connect to recipient first
int   iPort    = DEFAULT_PORT;          // Port to send data to
char  cChar    = DEFAULT_CHAR;          // Character to fill buffer
DWORD dwCount  = DEFAULT_COUNT,         // Number of messages to send
      dwLength = DEFAULT_BUFFER_LENGTH; // Length of buffer to send
char  szRecipient[128];                 // Recipient's IP or host name

//
// Function: usage
//
// Description:
//    Print usage information and exit
//
void usage()
{
    printf("usage: sender [-p:int] [-r:IP] "
           "[-c] [-n:x] [-b:x] [-d:c]\n\n");
    printf("       -p:int   Remote port\n");
    printf("       -r:IP    Recipient's IP address or host name\n");
    printf("       -c       Connect to remote IP first\n");
    printf("       -n:x     Number of times to send message\n");
    printf("       -b:x     Size of buffer to send\n");
    printf("       -d:c     Character to fill buffer with\n\n");
    ExitProcess(1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some global flags to
//    indicate what actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int i;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'p':        // Remote port
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 'r':        // Recipient's IP addr
```

```
                    if (strlen(argv[i]) > 3)
                        strcpy(szRecipient, &argv[i][3]);
                    break;
                case 'c':        // Connect to recipient's IP addr
                    bConnect = TRUE;
                    break;
                case 'n':        // Number of times to send message
                    if (strlen(argv[i]) > 3)
                        dwCount = atol(&argv[i][3]);
                    break;
                case 'b':        // Buffer size
                    if (strlen(argv[i]) > 3)
                        dwLength = atol(&argv[i][3]);
                    break;
                case 'd':        // Character to fill buffer
                    cChar = argv[i][3];
                    break;
                default:
                    usage();
                    break;
            }
        }
    }
}

//
// Function: main
//
// Description:
//    Main thread of execution. Initialize Winsock, parse the command
//    line arguments, create a socket, connect to the remote IP
//    address if specified, and then send datagram messages to the
//    recipient.
//
int main(int argc, char **argv)
{
    WSADATA        wsd;
    SOCKET         s;
    char          *sendbuf = NULL;
    int            ret,
                   i;
    SOCKADDR_IN    recipient;

    // Parse the command line and load Winsock
    //
    ValidateArgs(argc, argv);

    if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
    {
        printf("WSAStartup failed!\n");
        return 1;
    }
    // Create the socket
```

```
    //
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s == INVALID_SOCKET)
    {
        printf("socket() failed; %d\n", WSAGetLastError());
        return 1;
    }
    // Resolve the recipient's IP address or host name
    //
    recipient.sin_family = AF_INET;
    recipient.sin_port = htons((short)iPort);
    if ((recipient.sin_addr.s_addr = inet_addr(szRecipient))
        == INADDR_NONE)
    {
        struct hostent *host=NULL;

        host = gethostbyname(szRecipient);
        if (host)
            CopyMemory(&recipient.sin_addr, host->h_addr_list[0],
                host->h_length);
        else
        {
            printf("gethostbyname() failed: %d\n", WSAGetLastError());
            WSACleanup();
            return 1;
        }
    }
    // Allocate the send buffer
    //
    sendbuf = GlobalAlloc(GMEM_FIXED, dwLength);
    if (!sendbuf)
    {
        printf("GlobalAlloc() failed: %d\n", GetLastError());
        return 1;
    }
    memset(sendbuf, cChar, dwLength);
    //
    // If the connect option is set, "connect" to the recipient
    // and send the data with the send() function
    //
    if (bConnect)
    {
        if (connect(s, (SOCKADDR *)&recipient,
                sizeof(recipient)) == SOCKET_ERROR)
        {
            printf("connect() failed: %d\n", WSAGetLastError());
            GlobalFree(sendbuf);
            WSACleanup();
            return 1;
        }
        for(i = 0; i < dwCount; i++)
        {
            ret = send(s, sendbuf, dwLength, 0);
```

```
            if (ret == SOCKET_ERROR)
            {
                printf("send() failed: %d\n", WSAGetLastError());
                break;
            }
            else if (ret == 0)
                break;
            // Send() succeeded!
        }
    }
    else
    {
        // Otherwise, use the sendto() function
        //
        for(i = 0; i < dwCount; i++)
        {

            ret = sendto(s, sendbuf, dwLength, 0,
                    (SOCKADDR *)&recipient, sizeof(recipient));
            if (ret == SOCKET_ERROR)
            {
                printf("sendto() failed; %d\n", WSAGetLastError());
                break;
            }
            else if (ret == 0)
                break;
            // sendto() succeeded!
        }
    }
    closesocket(s);

    GlobalFree(sendbuf);
    WSACleanup();
    return 0;
}
```

Miscellaneous API Functions

In this section, we'll cover a few Winsock API functions that you might find useful when you put together your own network applications.

getpeername

This function is used to obtain the peer's socket address information on a connected socket. The function is defined as

```
int getpeername(
    SOCKET s,
    struct sockaddr FAR* name,
    int FAR* namelen
);
```

The first parameter is the socket for the connection, while the last two parameters are a pointer to a *SOCKADDR* structure of the underlying protocol type and its length. For datagram sockets, this function returns the address passed to a connect call; however, it will not return the address passed to a *sendto* or *WSASendTo* call.

getsockname

This function is the opposite of *getpeername*. It returns the address information for the local interface of a given socket. The function is defined as follows:

```
int getsockname(
    SOCKET s,
    struct sockaddr FAR* name,
    int FAR* namelen
);
```

The parameters are the same as the *getpeername* parameters except that the address information returned for socket *s* is the local address information. In the case of TCP, the address is the same as the server socket listening on a specific port and IP interface.

WSADuplicateSocket

The *WSADuplicateSocket* function is used to create a *WSAPROTOCOL_INFO* structure that can be passed to another process, thus enabling the other process to open a handle to the same underlying socket so that it too can perform operations on that resource. Note that this is only necessary between processes; threads in the same process can freely pass the socket descriptors. This function is defined as

```
int WSADuplicateSocket(
    SOCKET s,
    DWORD dwProcessId,
    LPWSAPROTOCOL_INFO lpProtocolInfo
);
```

The first parameter is the socket handle to duplicate. The second parameter, *dwProcessId*, is the process ID of the process that intends to use the duplicated socket. Third, the *lpProtocolInfo* parameter is a pointer to a *WSAPROTOCOL_INFO* structure that will contain the necessary information for the target process to open a duplicate handle. Some form of interprocess communication must occur so that the current process can pass the *WSAPROTOCOL_INFO* structure to the target process, which then uses this structure to create a handle to the socket (using the *WSASocket* function).

The descriptors in both sockets can be used independently for I/O; however, Winsock provides no access control, so it is up to the programmer to enforce some kind of synchronization. All of the state information associated with a socket is held in common across all the descriptors because the socket descriptors are duplicated, not the actual socket. For example, any socket option set by the *setsockopt* function on one of the descriptors is subsequently visible using the *getsockopt* function from any or all descriptors. If a process calls *closesocket* on a duplicated socket, it causes the descriptor in that process to become deallocated; however, the underlying socket will remain open until *closesocket* is called on the last remaining descriptor.

Additionally, be aware of some issues with notification on shared sockets when using *WSAAsyncSelect* and *WSAEventSelect*. These two functions are used in asynchronous I/O (discussed in Chapter 8). Issuing either of these calls using any of the shared descriptors cancels any previous event registration for the socket, regardless of which descriptor was used to make that registration. Thus, for example, a shared socket cannot deliver *FD_READ* events to process A and *FD_WRITE* events to process B. If you require event notifications on both descriptors, you should rethink the design of your application to use threads as opposed to processes.

TransmitFile

TransmitFile is a Microsoft-specific Winsock extension that allows for high-performance data transfers from a file. This is efficient is because the entire data transfer can occur in kernel mode. That is, if your application reads a chunk of data from the file and then uses *send* or *WSASend,* there are multiple send calls that involve user-mode-to-kernel-mode transitions. With *TransmitFile,* the entire read and send process is performed in kernel mode. The function is defined as

```
BOOL TransmitFile(
    SOCKET hSocket,
    HANDLE hFile,
    DWORD nNumberOfBytesToWrite,
    DWORD nNumberOfBytesPerSend,
    LPOVERLAPPED lpOverlapped,
    LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers,
    DWORD dwFlags
);
```

The *hSocket* parameter identifies the connected socket on which to transfer the file. The *hFile* parameter is a handle to an opened file. (This is the file that will be sent.) The *nNumberOfBytesToWrite* indicates how many bytes to write from the file. Passing 0 indicates the entire file should be sent. The *nNumberOfBytesPerSend* parameter indicates the send size to use for write operations. For example, specifying 2048 causes *TransmitFile* to send the given file on the socket in 2-KB chunks. Passing 0 indicates using the default send size. The *lpOverlapped* parameter specifies an *OVERLAPPED* structure that is used in overlapped I/O. (See Chapter 8 for information on overlapped I/O.)

The next parameter, *lpTransmitBuffers,* is a *TRANSMIT FILE BUFFERS* structure that contains data to be sent

before and after the file transfer. The structure is defined as

```
typedef struct _TRANSMIT_FILE_BUFFERS {
    PVOID Head;
    DWORD HeadLength;
    PVOID Tail;
    DWORD TailLength;
} TRANSMIT_FILE_BUFFERS;
```

The *Head* field is a pointer to the data to send before transmitting the file. *HeadLength* indicates the amount of data to send beforehand. The *Tail* field points to the data to send after the file is transmitted. *TailLength* is the number of bytes to send afterward.

The last parameter of *TransmitFile*, *dwFlags*, is used to specify flags to affect the behavior of *TransmitFile*. Table 7-2 contains the flags and their explanations.

Table 7-2. *TransmitFile flags*

Flag	*Description*
TF_DISCONNECT	Initiates socket closure after data has been sent.
TF_REUSE_SOCKET	Allows the socket handle to be reused in *AcceptEx* as a client socket.
TF_USE_DEFAULT_WORKER	Indicates that the transfer should take place in the context of the system's default thread. This is useful for long file transfers.
TF_USE_SYSTEM_THREAD	Indicates that the transfer should take place in the context of the system thread. This is also useful for long file transfers.
TF_USE_KERNEL_APC	Indicates that kernel Asynchronous Procedure Calls (APC) should process the file transfer. This can offer a significant performance increase if reading the file into the cache requires only one read.
TF_WRITE_BEHIND	Indicates that *TransmitFile* should complete without having all the data acknowledged by the remote system.

Windows CE

All the information in the preceding sections applies equally to Windows CE. The only exception is that because Windows CE is based on the Winsock 1.1 specification, none of the Winsock 2-specific functions—such as *WSA* variants of the sending, receiving, connecting, and accepting functions—is available. The only *WSA* functions available on Windows CE are *WSAStartup*, *WSACleanup*, *WSAGetLastError*, and *WSAIoctl*. We have already discussed the first three of these functions; the last will be covered in Chapter 9.

Windows CE supports the TCP/IP protocol, which means you have access to both TCP and UDP. In addition to TCP/IP, infrared sockets are also supported. The IrDA protocol supports only stream-oriented communication. For both protocols, you make all the usual Winsock 1.1 API calls for creating and transmitting data. The only exception has to do with a bug in UDP datagram sockets in Windows CE 2.0: every call to *send* or *sendto* causes a kernel memory leak. This bug was fixed in Windows CE 2.1, but because the kernel is distributed in ROM, no software updates can be distributed to fix the problem with Windows CE 2.0. The only solution is to avoid using datagrams in Windows CE 2.0.

Because Windows CE does not support console applications and uses UNICODE only, the examples presented in this chapter are targeted to Windows 95 and Windows 98, Windows NT, and Windows 2000. The purpose of our examples is to teach the core concepts of Winsock without having to trudge through code that doesn't relate to Winsock. Unless you're writing a service for Windows CE, a user interface is almost always required. This entails writing many additional functions for window handlers and other user-interface elements, which can obfuscate what we're trying to teach. Additionally, there is the dilemma of UNICODE vs. the non-UNICODE Winsock functions. It is up to the programmer to decide whether the strings passed to the sending and receiving Winsock functions are UNICODE or ANSI strings. Winsock doesn't care what you pass as long it's a valid buffer. (Of course, you might need to typecast the buffer to silence the compiler warnings.) Don't forget that if you cast a UNICODE string to *char *,* the length parameter for how many bytes to send should be adjusted accordingly! In Windows CE, if you want to display any data sent or received, you must take into account whether it is UNICODE so that it can be displayed, as all the other Win32 functions do require UNICODE strings. In sum, Windows CE requires a great deal more housekeeping to make a simple Winsock application.

If you do want to run these examples on Windows CE, only a few minor modifications are required for the Winsock code to compile. First the header file must be Winsock.h, as opposed to Winsock2.h. *WSAStartup* should load version 1.1 because that is the current version of Winsock in Windows CE. Also, Windows CE does not support console applications; therefore, you must use *WinMain* instead of *main*. Note that this does not mean you are required to incorporate a window into your application; it just means you can't use console text I/O functions such as *printf*.

Other Address Families

All the Winsock API functions introduced in this chapter are protocol-independent. That is, the usage presented here can easily be applied to the other protocols supported by Win32 platforms. The following sections merely describe the sample client/server code for the other protocol families found on the companion CD-ROM.

AppleTalk

A single AppleTalk sample is provided to illustrate basic client/server techniques. The sample supports both the AppleTalk PAP and ADSP protocols. The PAP protocol is a message-oriented, connectionless, unreliable protocol similar to UDP, but with two notable exceptions. First it supports partial messages, which means that a call to *WSARecvEx* will possibly return with only part of a datagram message. You must check for the *MSG_PARTIAL* flag on return to see whether additional calls are required to obtain the full message. The second exception is that you must set a socket option specific to the PAP protocol before every read. The option, *SO_PRIME_READ*, which is used with the *setsockopt* function, is discussed in Chapter 9. Take a look at the *Atalk.c* sample on the CD, which illustrates how to check for the *MSG_PARTIAL* flag and how to use the *SO_PRIME_READ* option.

The ADSP protocol is a connection-oriented, streaming, reliable protocol— much like TCP. The basic API calls for AppleTalk remain similar to the ones in the UDP and TCP examples presented in this chapter. The only differences will be specific to name resolution. Remember that for AppleTalk, you must bind to an empty address first before looking up or registering an AppleTalk name. This is discussed in more detail in the AppleTalk addressing section in Chapter 6.

The AppleTalk protocol has one limitation. Support for AppleTalk originated in Winsock 1.1, and when Winsock 2 was developed, it appears that AppleTalk was not fully "hooked" into the new functions. Using any of the *WSASend* or *WSARecv* functions might result in flaky results, such as negative byte count returns. This problem is actually described in the Knowledge Base article Q164565. The only exception is *WSARecvEx*, which is simply a *recv* call except that the *flags* parameter is in/out and can be queried for the *MSG_PARTIAL* flag upon return.

IrDA

The infrared protocol is a recent addition that is available on Windows CE, Windows 98, and Windows 2000. It offers only one protocol type, which is a connection-oriented, streaming, reliable protocol. Again, the only major difference in the code is the name resolution, which is significantly different from name resolution in IP. You should be aware of one other difference: because Windows CE supports only the Winsock 1.1 specification, you can use only Winsock 1.1 functions on infrared sockets on a Windows CE platform. On Windows 98 and Windows 2000, you can also use the functions specific to Winsock 2. The sample code uses only Winsock 1.1 functions. Of course, on Windows 98 and Windows 2000 you must load the Winsock 2.2 library or greater, as the support for the *AF_IRDA* address family is not available in earlier versions.

The sample code for infrared sockets is found in the following files: *Ircommon.h*, *Ircommon.c*, *Irclient.c*, and *Irserver.c*. The first two files simply define two common functions, one for sending data and the other for receiving data, which are used by both the client and the server. The client side is detailed in *Irclient.c*, which is straightforward. First all devices in range are enumerated. Then a connection attempt is made to each one with the given service name. The first device to accept the connection request is taken. Subsequently the client sends data and reads it back. On the server side of the equation is the file *Irserver.c*. The server simply creates an infrared socket, binds the specified service name to the socket, and waits for client connections. For each client, a thread is spawned to receive data and send it back to the client.

Note that these examples are written with Windows 98 and Windows 2000 in mind. Like the TCP/IP samples, these examples require only slight modifications to run on Windows CE. Regarding Windows CE, the two main points are that there is no support for console applications, and all functions (other than Winsock) use UNICODE strings.

NetBIOS

We've presented several Winsock NetBIOS examples. As you learned in Chapter 1, NetBIOS is capable of using several different transports, which is still the case with Winsock. In Chapter 6, you learned how to enumerate the NetBIOS-capable transports and how to create sockets based on any one of these. Each protocol-to-adapter combination has two entries: one *SOCK_DGRAM* and one *SOCK_SEQPACKET* type. These correspond to

connectionless datagram and stream sockets that are quite similar to UDP and TCP sockets. Besides name resolution, the NetBIOS Winsock interface is no different from what is presented earlier in this chapter. Remember that a well-written server should listen on all available LANAs and that the client should attempt to connect on all LANAs on its end.

The first examples on the CD are *Wsnbsvr.c* and *Wsnbclnt.c*. These examples use the *SOCK_SEQPACKET* socket type, which for all practical purposes appears to the programmer to be stream-oriented. The server creates a socket for each LANA, which is enumerated with the *WSAEnumProtocols* function, and binds it to the server's well-known name. Once a client connection is made, the server creates a thread to handle the connection. From there, the thread simply reads incoming data and echoes it back to the client. Similarly, the client attempts to connect on all LANAs. After the first connect succeeds, the other sockets are closed. The client then sends data to the server and the server reads it back.

The other example is *Wsnbdgs.c*, which is a datagram, or *SOCK_DGRAM*, example. This example includes the code for both sending and receiving datagram messages. This is a connectionless protocol, so the message sent to the server is sent on all available transports (since you really don't have any idea beforehand which transport or transports will be able to reach the server). Additionally, this example supports unique, group, and broadcast data (all discussed in Chapter 1).

IPX/SPX

The IPX/SPX example, *Sockspx.c*, illustrates how to use the IPX protocol as well as stream and sequential packet SPXII. This single sample incorporates both the sender and the receiver for all three protocols. The particular protocol used is specified via the -p command line option. The sample is straightforward and easy to follow. The main function parses the command line arguments and then calls either the *Server* or the *Client* function. For the connection-oriented SPXII protocol, this means that the server binds the socket to the internal network address and waits for client connections while the client attempts to connect to the server that is specified on the command line. Once the connection is established, data is sent and received in the normal fashion.

For the connectionless IPX protocol, the example is even simpler. The server simply binds to the internal network and waits for incoming data by calling the *recvfrom* function. The client sends data to the recipient specified on the command line via the *sendto* function.

Two sections of the example might need a bit of explanation. First is the function *FillIpxAddress*, which is responsible for encoding an ASCII IPX address specified on the command line into a *SOCKADDR_IPX* structure. As you saw in Chapter 6, IPX represents its addresses as hexadecimal strings, which means that each hexadecimal character in the address actually occupies 4 bits within the various address fields of the *SOCKADDR_IPX* structure. *FillIpxAddress* takes the IPX address and calls another function, *AtoH*, which actually performs the conversion.

The second function that needs explanation is *EnumerateAdapters,* which is executed if the -m flag is given on the command line. This function uses the socket option *IPX_MAX_ADAPTER_NUM* to find out how many local IPX addresses are available and then calls the *IPX_ADDRESS* socket option to obtain each address. These socket options and their parameters are discussed in Chapter 9. We use these options because our example uses straight IPX addresses and does not perform any name resolution. Chapter 10 examines the name registration and name resolution that are possible for IPX.

ATM

The ATM protocol is accessible from Winsock on Windows 98 and Windows 2000. The ATM sample is contained in the files *Wsockatm.c*, *Support.c*, and *Support.h*. The latter two files simply contain support routines used by *Wsockatm.c*, such as local ATM address enumeration and ATM address encoding. ATM addresses are hexadecimal encoded, just like IPX addresses, and we use the same *AtoH* function. We also use the socket ioctl command *SIO_GET_NUMBER_OF_ATM_DEVICES* to get the number of local ATM interfaces and then use the ioctl command *SIO_GET_ATM_ADDRESS* to retrieve the actual address. These ioctl commands are covered in Chapter 9.

Otherwise, both the client and server sides are implemented within *Wsockatm.c*. Because ATM supports only connection-oriented communication, the sample isn't very long and the majority of the code is given in the *main* function. The server will bind to an explicit local interface and wait for client connections, which are handled in the same thread as the listening socket. This means the server will only be able to service one client at a time. We designed the sample this way on purpose, to keep the code simple. On the other hand, the client calls *connect* with the server's ATM address. Once the connection is established, data is sent on the connection.

A few words of caution when using the ATM protocol: you will notice that after the *WSAAccept* call is made within the server, the address of the client is printed out. However, at the time the server receives the connection

request, the client's address is not known. This is because the connection is not fully established when the *accept* function is triggered. This is also true on the client side. When the client makes a call to connect to the server, it will succeed even though the connection has not been fully established. This means that upon completion of a *connect* or an *accept* call, an attempt to send data immediately might silently fail until the connection is fully established. Unfortunately, there is no way for the application to determine at what point the connection becomes valid. Additionally, ATM supports only hard closes. That is, when the application calls *closesocket*, the connection is immediately terminated. For protocols that do not support graceful close, any data pending on the socket at either end is normally discarded at the point *closesocket* is called. This is perfectly acceptable behavior; however, the ATM provider is nice to developers. When data is pending on the socket and one party has closed its socket, Winsock still returns the data queued for receiving on the socket.

Conclusion

In Chapter 6, you learned how to create a socket for a given protocol and how to resolve a host name for the protocol's address family. In this chapter, we took that knowledge and presented the basic Winsock functions that are required for those connection-oriented and connectionless protocols. For connection-oriented protocols, you know how to accept a client connection and how to establish a client connection to a server. We covered the semantics for session-oriented data-send operations and data-receive operations. For connectionless protocols, you also learned how to send and receive data. Of course, we presented this information using only one I/O model: blocking sockets. In the next chapter, we will cover the other models available in Winsock that make it such a powerful API.

Chapter 8

Winsock I/O Methods

This chapter focuses on managing I/O in a Windows sockets application. Winsock features socket modes and socket I/O models to control how I/O is processed on a socket. A socket *mode* simply determines how Winsock functions behave when called with a socket. A socket *model,* on the other hand, describes how an application manages and processes I/O on a socket. Socket I/O models are independent of socket modes. You will find that socket models are available to overcome the limitations of socket modes.

Winsock features two socket modes: *blocking* and *nonblocking*. The first part of this chapter describes these modes in detail and demonstrates how an application can use them to manage I/O. As you'll see later in the chapter, Winsock offers some interesting I/O models that help applications manage communication on one or more sockets at a time in an asynchronous fashion: *select, WSAAsyncSelect*, *WSAEventSelect*, *overlapped I/O,* and *completion port*. By the chapter's end, we'll review the pros and cons of the various socket modes and I/O models and help you decide which one best meets your application's needs.

All Windows platforms offer blocking and nonblocking socket operating modes. However, not every I/O model is available for every platform. As you can see in Table 8-1, only one of the I/O models is available under current versions of Windows CE. Windows 98 and Windows 95 (depending on whether you have Winsock version 1 or 2) support most of the I/O models with the exception of I/O completion ports. Every I/O model is available on Windows NT and Windows 2000.

Table 8-1. *Available socket I/O models*

Platform	*select*	*WSAAsyncSelect*	*WSAEventSelect*	Overlapped	Completion Port
Windows CE	Yes	No	No	No	No
Windows 95 (Winsock 1)	Yes	Yes	No	No	No
Windows 95 (Winsock 2)	Yes	Yes	Yes	Yes	No
Windows 98	Yes	Yes	Yes	Yes	No
Windows NT	Yes	Yes	Yes	Yes	Yes
Windows 2000	Yes	Yes	Yes	Yes	Yes

Socket Modes

As we mentioned earlier, Windows sockets perform I/O operations in two socket operating modes: blocking and nonblocking. In blocking mode, Winsock calls that perform I/O—such as *send* and *recv*—wait until the operation is complete before they return to the program. In nonblocking mode, the Winsock functions return immediately. Applications running on the Windows CE and Windows 95 (with Winsock 1) platforms, which support very few of the I/O models, require you to take certain steps with blocking and nonblocking sockets to handle a variety of situations.

Blocking Mode

Blocking sockets cause concern because any Winsock API call on a blocking socket can do just that—block for some period of time. Most Winsock applications follow a producer-consumer model in which the application reads (or writes) a specified number of bytes and performs some computation on that data. The code snippet in Figure 8-1 illustrates this model.

Figure 8-1. *Simple blocking socket sample*

```
SOCKET  sock;
char    buff[256];
int     done = 0;

...

while(!done)
{
    nBytes = recv(sock, buff, 65);
    if (nBytes == SOCKET_ERROR)

    {
        printf("recv failed with error %d\n",
            WSAGetLastError());
        Return;
    }
    DoComputationOnData(buff);
}

...
```

The problem with this code is that the *recv* function might never return if no data is pending because the statement says to return only after reading some bytes from the system's input buffer. Some programmers might be tempted to peek for the necessary number of bytes in the system's buffer by using the *MSG_PEEK* flag in *recv* or by calling *ioctlsocket* with the *FIONREAD* option. Peeking for data without actually reading the data (reading the data actually removes it from the system's buffer) is considered bad programming practice and should be avoided at all costs. The overhead associated with peeking is great because one or more system calls are necessary just to check the number of bytes available. Then, of course, there is the overhead of making the actual *recv* call that removes the data from the system buffer. What can be done to avoid this? The idea is to prevent the application from totally freezing because of lack of data (either from network problems or from client problems) without continually peeking at the system network buffers. One method is to separate the application into a reading thread and a computation thread. Both threads share a common data buffer. Access to this buffer is protected through the use of a synchronization object, such as an event or a mutex. The purpose of the reading thread is to continually read data from the network and place it in the shared buffer. When the reading thread has read the minimum amount of data necessary for the computation thread to do its work, it can signal an event that notifies

the computation thread to begin. The computation thread then removes a chunk of data from the buffer and performs the necessary calculations.

Figure 8-2 illustrates this approach by providing two functions, one responsible for reading network data (*ReadThread*) and one for performing the computations on the data (*ProcessThread*).

Figure 8-2. *Multithreaded blocking sockets example*

```
// Initialize critical section (data) and create
// an auto-reset event (hEvent) before creating the
// two threads
CRITICAL_SECTION data;
HANDLE           hEvent;
TCHAR            buff[MAX_BUFFER_SIZE];
int              nbytes;

...

// Reader thread
void ReadThread(void)
{
    int nTotal = 0,
        nRead = 0,
        nLeft = 0,
        nBytes = 0;

    while (!done)
    {
        nTotal = 0;
        nLeft = NUM_BYTES_REQUIRED;
        while (nTotal != NUM_BYTES_REQUIRED)
        {
            EnterCriticalSection(&data);
            nRead = recv(sock, &(buff[MAX_BUFFER_SIZE - nBytes]),
                nLeft);
            if (nRead == -1)
            {
                printf("error\n");
                ExitThread();
            }
            nTotal += nRead;
            nLeft -= nRead;

            nBytes += nRead;
            LeaveCriticalSection(&data);
        }
        SetEvent(hEvent);
    }
}

// Computation thread
void ProcessThread(void)
{
```

```
    WaitForSingleObject(hEvent);

    EnterCriticalSection(&data);
    DoSomeComputationOnData(buff);

    // Remove the processed data from the input
    // buffer, and shift the remaining data to
    // the start of the array
    nBytes -= NUM_BYTES_REQUIRED;

    LeaveCriticalSection(&data);
}
```

One drawback of blocking sockets is that communicating via more than one connected socket at a time becomes difficult for the application. Using the foregoing scheme, the application could be modified to have a reading thread and a data processing thread per connected socket. This adds quite a bit of housekeeping overhead, but it is a feasible solution. The only drawback is that the solution does not scale well once you start dealing with a large number of sockets.

Nonblocking Mode

The alternative to blocking sockets is nonblocking sockets. Nonblocking sockets are a bit more challenging to use, but they are every bit as powerful as blocking sockets, with a few advantages. Figure 8-3 illustrates how to create a socket and put it into nonblocking mode.

Figure 8-3. *Making a socket nonblocking*

```
SOCKET        s;
unsigned long ul = 1;
int           nRet;

s = socket(AF_INET, SOCK_STREAM, 0);
nRet = ioctlsocket(s, FIOBIO, (unsigned long *) &ul);
if (nRet == SOCKET_ERROR)
{
    // Failed to put the socket into nonblocking mode
}
```

Once a socket is placed in nonblocking mode, Winsock API calls return immediately. In most cases, these calls fail with the error *WSAEWOULDBLOCK*, which means that the requested operation did not have time to complete during the call. For example, a call to *recv* returns *WSAEWOULDBLOCK* if no data is pending in the system's input buffer. Often additional calls to the same function are required until a successful return code is encountered. Table 8-2 describes the meaning of *WSAEWOULDBLOCK* when returned by commonly used Winsock calls.

Because nonblocking calls frequently fail with the *WSAEWOULDBLOCK* error, you should check all return codes and be prepared for failure at any time. The pitfall many programmers fall into is that of continually calling a function until it returns a success. For example, placing a call to *recv* in a tight loop to read 200 bytes of data is no better than polling a blocking socket with the *MSG_PEEK* flag mentioned earlier. Winsock's socket I/O models can help an application determine when a socket is available for reading and writing.

Table 8-2. WSAEWOULDBLOCK *errors on nonblocking sockets*

Function Name	Description
WSAAccept and *accept*	The application has not received a connection request. Call again to check for a connection.
closesocket	In most cases, this means that *setsockopt* was called with the *SO_LINGER* option and a nonzero timeout was set.
WSAConnect and *connect*	The connection is initiated. Call again to check for completion.
WSARecv, *recv*, *WSARecvFrom* and *recvfrom*	No data has been received. Check again later.
WSASend, *send*, *WSASendTo*, and *sendto*	No buffer space available for outgoing data. Try again later.

Each socket mode—blocking and nonblocking—has advantages and disadvantages. Blocking sockets are easier to use from a conceptual standpoint but become difficult to manage when dealing with multiple connected sockets or when data is sent and received in varying amounts and at arbitrary times. On the other hand, nonblocking sockets are more difficult in the sense that more code needs to be written to handle the possibility of receiving a *WSAEWOULDBLOCK* error on every Winsock call. Socket I/O models help applications manage communications on one or more sockets at a time in an asynchronous fashion.

Socket I/O Models

Essentially five types of socket I/O models are available that allow Winsock applications to manage I/O: *select, WSAAsyncSelect*, *WSAEventSelect*, *overlapped,* and *completion port.* This section explains the features of each I/O model and outlines how to use the model to develop an application that can manage one or more socket requests. On the companion CD, you will find one or more sample applications for each I/O model demonstrating how to develop a simple TCP echo server using the principles described in each model.

The *select* Model

The *select* model is the most widely available I/O model in Winsock. We call it the *select* model because it centers on using the *select* function to manage I/O. The design of this model originated on Unix-based computers featuring Berkeley socket implementations. The *select* model was incorporated into Winsock 1.1 to allow applications that want to avoid blocking on socket calls the ability to manage multiple sockets in an organized manner. Because Winsock 1.1 is backward-compatible with Berkeley socket implementations, a Berkeley socket application that uses the *select* function should technically be able to run without modification.

The *select* function can be used to determine whether there is data on a socket and whether a socket can be written to. The whole reason for having this function is to prevent your application from blocking on an I/O bound call such as *send* or *recv* when a socket is in a blocking mode and to prevent the *WSAEWOULDBLOCK* error when a socket is in nonblocking mode. The *select* function blocks for I/O operations until the conditions specified as parameters are met. The function prototype for *select* is as follows:

```
int select(
    int nfds,
    fd_set FAR * readfds,
    fd_set FAR * writefds,
    fd_set FAR * exceptfds,
    const struct timeval FAR * timeout
);
```

The first parameter, *nfds*, is ignored and is included only for compatibility with Berkeley socket applications. You'll notice that there are three *fd_set* parameters: one for checking readability (*readfds*), one for writability (*writefds*), and one for out-of-band data (*exceptfds*). Essentially, the *fd_set* data type represents a collection of sockets. The *readfds* set identifies sockets that meet one of the following conditions:

- Data is available for reading.
- Connection has been closed, reset, or terminated.
- If *listen* has been called and a connection is pending, the *accept* function will succeed.

The *writefds* set identifies sockets in which one of the following is true:

- Data can be sent.
- If a nonblocking connect call is being processed, the connection has succeeded.

Finally, the *exceptfds* set identifies sockets in which one of the following is true:

- If a nonblocking connect call is being processed, the connection attempt failed.
- Out-of-band (OOB) data is available for reading.

For example, when you want to test a socket for readability, you must add your socket to the *readfds* set and wait for the *select* function to complete. When the *select* call completes, you have to determine whether your socket is

still part of the *readfds* set. If so, the socket is readable—you can begin to retrieve data from the socket. Any two of the three parameters (*readfds*, *writefds*, *exceptfds*) can be null values (at least one must not be null), and any non-null set must contain at least one socket handle; otherwise, the *select* function won't have anything to wait for. The final parameter, *timeout*, is a pointer to a *timeval* structure that determines how long the *select* function will wait for I/O to complete. If *timeout* is a null pointer, *select* will block indefinitely until at least one descriptor meets the specified criteria. The *timeval* structure is defined as

```
struct timeval
{
    long tv_sec;
    long tv_usec;
};
```

The *tv_sec* field indicates how long to wait in seconds; the *tv_usec* field indicates how long to wait in milliseconds. The timeout value {0, 0} indicates *select* will return immediately, allowing an application to poll on the *select* operation. This should be avoided for performance reasons. When *select* completes successfully, it returns the total number of socket handles that have I/O operations pending in the *fd_set* structures. If the *timeval* limit expires, it returns 0. If *select* fails for any reason, it returns *SOCKET_ERROR*.

Before you can begin to use *select* to monitor sockets, your application has to set up either one or all of the read, write, and exception *fd_set* structures by assigning socket handles to a set. When you assign a socket to one of the sets, you are asking *select* to let you know whether the I/O activities described above have occurred on a socket. Winsock provides the following set of macros to manipulate and check the *fd_set* sets for I/O activity.

- *FD_CLR(s, *set)* Removes socket *s* from *set*
- *FD_ISSET(s, *set)* Checks to see whether *s* is a member of *set* and returns *TRUE* if so
- *FD_SET(s, *set)* Adds socket *s* to *set*
- *FD_ZERO(*set)* Initializes *set* to the empty set

For example, if you want to find out when it is safe to read data from a socket without blocking, simply assign your socket to the *fd_read* set using the *FD_SET* macro and then call *select*. To test whether your socket is still part of the *fd_read* set, use the *FD_ISSET* macro. The following steps describe the basic flow of an application that uses *select* with one or more socket handles:

1. Initialize each *fd_set* of interest, using the *FD_ZERO* macro.
2. Assign socket handles to each of the *fd_set* sets of interest, using the *FD_SET* macro.
3. Call the *select* function, and wait until I/O activity sets one or more of the socket handles in each *fd_set* set provided. When *select* completes, it returns the total number of socket handles that are set in all of the *fd_set* sets and updates each set accordingly.
4. Using the return value of *select*, your application can determine which application sockets have I/O pending by checking each *fd_set* set using the *FD_ISSET* macro.
5. After determining which sockets have I/O pending in each of the sets, process the I/O and go to step 1 to continue the *select* process.

When select returns, it modifies each of the *fd_set* structures by removing the socket handles that do not have pending I/O operations. This is why you should use the *FD_ISSET* macro as in step 4 above to determine whether a particular socket is part of a set. Figure 8-4 outlines the basic steps needed to set up the select model for a single socket. Adding more sockets to this application simply involves maintaining a list or an array of additional sockets.

Figure 8-4. *Managing I/O on a socket using* select

```
SOCKET  s;
fd_set  fdread;
int     ret;

// Create a socket, and accept a connection

// Manage I/O on the socket
while(TRUE)
{
    // Always clear the read set before calling
    // select()
    FD_ZERO(&fdread);

    // Add socket s to the read set
    FD_SET(s, &fdread);

    if ((ret = select(0, &fdread, NULL, NULL, NULL))
        == SOCKET_ERROR)
    {

        // Error condition
    }

    if (ret > 0)
    {
        // For this simple case, select() should return
        // the value 1. An application dealing with
        // more than one socket could get a value
        // greater than 1. At this point, your
        // application should check to see whether the
        // socket is part of a set.

        if (FD_ISSET(s, &fdread))
        {
            // A read event has occurred on socket s
        }
    }
}
```

The *WSAAsyncSelect* Model

Winsock provides a useful asynchronous I/O model that allows an application to receive Windows message-based notification of network events on a socket. This is accomplished by calling the *WSAAsyncSelect* function after creating a socket. This model originally existed in Winsock 1.1 implementations to help application programmers cope with the cooperative multitasking message-based environment of 16-bit Windows platforms, such as Windows for Workgroups. Applications can still benefit from this model, especially if they manage window messages in a standard Windows procedure, normally referred to as a *winproc*. This model is also used by the Microsoft Foundation Class (MFC) *CSocket* object.

Message notification

To use the *WSAAsyncSelect* model, your application must first create a window using the *CreateWindow* function and supply a window procedure (*winproc*) support function for this window. You can also use a dialog box with a dialog procedure instead of a window because dialog boxes *are* windows. For our purposes, we will demonstrate this model using a simple window with a supporting window procedure. Once you have set up the window infrastructure, you can begin creating sockets and turning on window message notification by calling the *WSAAsyncSelect* function, which is defined as

```
int WSAAsyncSelect(
    SOCKET s,
    HWND hWnd,
    unsigned int wMsg,
    long lEvent
);
```

The *s* parameter represents the socket we are interested in. The *hWnd* parameter is a window handle identifying the window or the dialog box that receives a message when a network event occurs. The *wMsg* parameter identifies the message to be received when a network event occurs. This message is posted to the window that is identified by the *hWnd* window handle. Normally applications set this message to a value greater than the Windows *WM_USER* value to avoid confusing a network window message with a predefined standard window message. The last parameter, *lEvent*, represents a bitmask that specifies a combination of network events—listed in Table 8-3—that the application is interested in. Most applications are typically interested in the *FD_READ*, *FD_WRITE*, *FD_ACCEPT*, *FD_CONNECT*, and *FD_CLOSE* network event types. Of course, the use of the *FD_ACCEPT* or the *FD_CONNECT* type depends on whether your application is a client or a server. If your application is interested in more than one network event, simply set this field by performing a bitwise OR on the types and assigning them to *lEvent*. For example:

```
WSAAsyncSelect(s, hwnd, WM_SOCKET,
    FD_CONNECT | FD_READ | FD_WRITE | FD_CLOSE);
```

This allows our application to get connect, send, receive, and socket-closure network event notifications on socket *s.* It is impossible to register multiple events one at a time on the socket. Also note that once you turn on event notification on a socket, it remains on unless the socket is closed by a call to *closesocket* or the application changes the registered network event types by calling *WSAAsyncSelect* (again, on the socket). Setting the *lEvent* parameter to 0 effectively stops all network event notification on the socket.

When your application calls *WSAAsyncSelect* on a socket, the socket mode is automatically changed from blocking to the nonblocking mode that we described earlier. As a result, if a Winsock I/O call such as *WSARecv* is called and has to wait for data, it will fail with error *WSAEWOULDBLOCK*. To avoid this error, applications should rely on the user-defined window message specified in the *wMsg* parameter of *WSAAsyncSelect* to indicate when network event types occur on the socket.

Table 8-3. *Network event types for the* WSAAsyncSelect *function*

Event Type	Meaning
FD_READ	The application wants to receive notification of readiness for reading.
FD_WRITE	The application wants to receive notification of readiness for writing.
FD_OOB	The application wants to receive notification of the arrival of out-of-band (OOB) data.
FD_ACCEPT	The application wants to receive notification of incoming connections.
FD_CONNECT	The application wants to receive notification of a completed connection or a multipoint *join* operation.
FD_CLOSE	The application wants to receive notification of socket closure.
FD_QOS	The application wants to receive notification of socket Quality of Service (QOS) changes.
FD_GROUP_QOS	The application wants to receive notification of socket group Quality of Service (QOS) changes (reserved for future use with socket groups).
FD_ROUTING_INTERFACE_CHANGE	The application wants to receive notification of routing interface changes for the specified destination(s).
FD_ADDRESS_LIST_CHANGE	The application wants to receive notification of local address list changes for the socket's protocol family.

After your application successfully calls *WSAAsyncSelect* on a socket, the application begins to receive network event notification as Windows messages in the window procedure associated with the *hWnd* parameter window handle. A window procedure is normally defined as

```
LRESULT CALLBACK WindowProc(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam
);
```

The *hWnd* parameter is a handle to the window that invoked the window procedure. The *uMsg* parameter indicates which message needs to be processed. In our case, we will be looking for the message defined in the *WSAAsyncSelect* call. The *wParam* parameter identifies the socket on which a network event has occurred. This is important if you have more than one socket assigned to this window procedure. The *lParam* parameter contains two important pieces of information—the low word of *lParam* specifies the network event that has occurred, and the high word of *lParam* contains any error code.

When network event messages arrive at a window procedure, the application should first check the *lParam* high-word bits to determine whether a network error has occurred on the socket. There is a special macro, *WSAGETSELECTERROR*, which returns the value of the high-word bits error information. After the application has verified that no error occurred on the socket, the application should determine which network event type caused the Windows message to fire by reading the low-word bits of *lParam*. Another special macro, *WSAGETSELECTEVENT*, returns the value of the low-word portion of *lParam*.

Figure 8-5 demonstrates how to manage window messages when using the *WSAAsyncSelect* I/O model. The figure highlights the steps needed to develop a basic server application and removes the programming details of developing a fully featured Windows application.

Figure 8-5. WSAAsyncSelect *server sample code*

```
#define WM_SOCKET WM_USER + 1
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance, LPSTR lpCmdLine,
    int nCmdShow)
{
    SOCKET Listen;
    HWND Window;

    // Create a window and assign the ServerWinProc
    // below to it

    Window = CreateWindow();
    // Start Winsock and create a socket

    WSAStartup(...);
    Listen = Socket();

    // Bind the socket to port 5150
    // and begin listening for connections

    InternetAddr.sin_family = AF_INET;
    InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    InternetAddr.sin_port = htons(5150);

    bind(Listen, (PSOCKADDR) &InternetAddr,
        sizeof(InternetAddr));

    // Set up window message notification on
    // the new socket using the WM_SOCKET define
    // above

    WSAAsyncSelect(Listen, Window, WM_SOCKET,
        FD_ACCEPT | FD_CLOSE);

    listen(Listen, 5);

    // Translate and dispatch window messages
    // until the application terminates
}

BOOL CALLBACK ServerWinProc(HWND hDlg,WORD wMsg,
    WORD wParam, DWORD lParam)
{
    SOCKET Accept;

    switch(wMsg)
    {
        case WM_PAINT:
            // Process window paint messages
            break;
```

```
        case WM_SOCKET:

            // Determine whether an error occurred on the
            // socket by using the WSAGETSELECTERROR() macro

            if (WSAGETSELECTERROR(lParam))
            {
                // Display the error and close the socket
                closesocket(wParam);
                break;
            }

            // Determine what event occurred on the
            // socket

            switch(WSAGETSELECTEVENT(lParam))
            {
                case FD_ACCEPT:

                    // Accept an incoming connection
                    Accept = accept(wParam, NULL, NULL);

                    // Prepare accepted socket for read,
                    // write, and close notification

                    WSAAsyncSelect(Accept, hwnd, WM_SOCKET,
                        FD_READ | FD_WRITE | FD_CLOSE);
                    break;

                case FD_READ:

                    // Receive data from the socket in
                    // wParam
                    break;

                case FD_WRITE:

                    // The socket in wParam is ready
                    // for sending data
                    break;

                case FD_CLOSE:

                    // The connection is now closed
                    closesocket(wParam);
                    break;
            }
            break;
    }
    return TRUE;
}
```

One final detail worth noting is how applications should process *FD_WRITE* event notifications. *FD_WRITE* notifications are sent under only three conditions:

- After a socket is first connected with connect or WSAConnect
- After a socket is accepted with accept or WSAAccept
- When a *send*, *WSASend*, *sendto*, or *WSASendTo* operation fails with *WSAEWOULDBLOCK* and buffer space becomes available

Therefore, an application should assume that sends are always possible on a socket starting from the first *FD_WRITE* message and lasting until a *send*, *WSASend*, *sendto*, or *WSASendTo* returns the socket error *WSAEWOULDBLOCK*. After such failure, another *FD_WRITE* message notifies the application that sends are once again possible.

The *WSAEventSelect* Model

Winsock provides another useful asynchronous I/O model that is similar to the *WSAAsyncSelect* model that allows an application to receive event-based notification of network events on one or more sockets. This model is similar to the *WSAAsyncSelect* model in that your application receives and processes the same network events listed in Table 8-3 that the *WSAAsyncSelect* model uses. The major difference with this model is that network events are posted to an event object handle instead of a window procedure.

Event notification

The event notification model requires your application to create an event object for each socket used by calling the *WSACreateEvent* function, which is defined as

```
WSAEVENT WSACreateEvent(void);
```

The *WSACreateEvent* function simply returns an event object handle. Once you have an event object handle, you have to associate it with a socket and register the network event types of interest, as shown in Table 8-3. This is accomplished by calling the *WSAEventSelect* function, which is defined as

```
int WSAEventSelect(
    SOCKET s,
    WSAEVENT hEventObject,
    long lNetworkEvents
);
```

The *s* parameter represents the socket of interest. The *hEventObject* parameter represents the event object—obtained with *WSACreateEvent*—to associate with the socket. The last parameter, *lNetworkEvents*, represents a bitmask that specifies a combination of network event types (listed in Table 8-3) that the application is interested in. For a detailed discussion of these event types, see the *WSAAsyncSelect* I/O model discussed earlier.

The event created for *WSAEventSelect* has two operating states and two operating modes. The operating states are known as *signaled* and *nonsignaled*. The operating modes are known as *manual reset* and *auto reset*. *WSACreateEvent* initially creates event handles in a nonsignaled operating state with a manual reset operating mode. As network events trigger an event object associated with a socket, the operating state changes from nonsignaled to signaled. Because the event object is created in a manual reset mode, your application is responsible for changing the operating state from signaled to nonsignaled after processing an I/O request. This can be accomplished by calling the *WSAResetEvent* function, which is defined as

```
BOOL WSAResetEvent(WSAEVENT hEvent);
```

The function takes an event handle as its only parameter and returns *TRUE* or *FALSE* based on the success or failure of the call. When an application is finished with an event object, it should call the *WSACloseEvent* function to free the system resources used by an event handle. The *WSACloseEvent* function is defined as

```
BOOL WSACloseEvent(WSAEVENT hEvent);
```

This function also takes an event handle as its only parameter and returns *TRUE* if successful or *FALSE* if the call fails.

Once a socket is associated with an event object handle, the application can begin processing I/O by waiting for network events to trigger the operating state of the event object handle. The *WSAWaitForMultipleEvents* function is designed to wait on one or more event object handles and returns either when one or all of the specified handles are in the signaled state or when a specified timeout interval expires. *WSAWaitForMultipleEvents* is defined as

```
DWORD WSAWaitForMultipleEvents(
    DWORD cEvents,
    const WSAEVENT FAR * lphEvents,
    BOOL fWaitAll,
    DWORD dwTimeout,
    BOOL fAlertable
);
```

The *cEvents* and *lphEvents* parameters define an array of *WSAEVENT* objects in which *cEvents* represents the number of event objects in the array and *lphEvents* is a pointer to the array. *WSAWaitForMultipleEvents* can support only a maximum of *WSA_MAXIMUM_WAIT_EVENTS* objects, which is defined as 64. Therefore, this I/O model is capable of supporting only a maximum of 64 sockets at a time for each thread that makes the *WSAWaitForMultipleEvents* call. If you need to have this model manage more than 64 sockets, you should create additional worker threads to wait on more event objects. The *fWaitAll* parameter specifies how *WSAWaitForMultipleEvents* waits for objects in the event array. If *TRUE*, the function returns when all event objects in the *lphEvents* array are signaled. If *FALSE*, the function returns when any one of the event objects is signaled. In the latter case, the return value indicates which event object caused the function to return. Typically, applications set this parameter to *FALSE* and service one socket event at a time. The *dwTimeout* parameter specifies how long (in milliseconds) *WSAWaitForMultipleEvents* will wait for a network event to occur. The function returns if the interval expires, even if conditions specified by the *fWaitAll* parameter are not satisfied. If the timeout value is 0, the function tests the state of the specified event objects and returns immediately, which effectively allows an application to poll on the event objects. Setting the timeout value to 0 should be avoided for performance reasons. If no events are ready for processing, *WSAWaitForMultipleEvents* returns *WSA_WAIT_TIMEOUT*. If *dwsTimeout* is set to *WSA_INFINITE*, the function returns only when a network event signals an event object. The final parameter, *fAlertable*, can be ignored when you're using the *WSAEventSelect* model and should be set to *FALSE*. It is intended for use in processing completion routines in the overlapped I/O model, which will be described later in this chapter.

When *WSAWaitForMultipleEvents* receives network event notification of an event object, it returns a value indicating the event object that caused the function to return. As a result, your application can determine which network event type is available on a particular socket by referencing the signaled event in the event array and matching it with the socket associated with the event. When you reference the events in the event array, you should reference them using the return value of *WSAWaitForMultipleEvents* minus the predefined value *WSA_WAIT_EVENT_0*. For example:

```
Index = WSAWaitForMultipleEvents(...);
MyEvent = EventArray[Index _ WSA_WAIT_EVENT_0];
```

Once you have the socket that caused the network event, you can determine which network events are available by calling the *WSAEnumNetworkEvents* function, which is defined as

```
int WSAEnumNetworkEvents(
    SOCKET s,
    WSAEVENT hEventObject,
    LPWSANETWORKEVENTS lpNetworkEvents
);
```

The *s* parameter represents the socket that caused the network event. The *hEventObject* parameter is an optional parameter representing an event handle identifying an associated event object to be reset. Since our event object is in a signaled state, we can pass it in and it will be set to a nonsignaled state. If you don't want to use the *hEventObject* parameter for resetting events, you can use the *WSAResetEvent* function, which we described earlier. The final parameter, *lpNetworkEvents*, takes a pointer to a *WSANETWORKEVENTS* structure, which is used to retrieve network event types that occurred on the socket and any associated error codes. The *WSANETWORKEVENTS* structure is defined as

```
typedef struct _WSANETWORKEVENTS
{
    long lNetworkEvents;
    int  iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

The *lNetworkEvents* parameter is a value that indicates all the network event types (see Table 8-3) that have occurred on the socket.

> NOTE
>
> More than one network event type can occur when an event is signaled. For example, a busy server application might receive *FD_READ* and *FD_WRITE* notification at the same time.

The *iErrorCode* parameter is an array of error codes that are associated with the events in *lNetworkEvents*. For each network event type, there exists a special event index similar to the event type names—except for an additional "_BIT" string appended to the event name. For example, for the *FD_READ* event type, the index identifier for the *iErrorCode* array is named *FD_READ_BIT*. The following code fragment demonstrates this for an *FD_READ* event:

```
// Process FD_READ notification
if (NetworkEvents.lNetworkEvents & FD_READ)
{
    if (NetworkEvents.iErrorCode[FD_READ_BIT] != 0)
    {
       printf("FD_READ failed with error %d\n",
           NetworkEvents.iErrorCode[FD_READ_BIT]);
    }
}
```

After you process the events in the *WSANETWORKEVENTS* structure, your application should continue waiting for more network events on all of the available sockets. Figure 8-6 demonstrates how to develop a server and manage event objects when using the *WSAEventSelect* I/O model. The figure highlights the steps needed to develop a basic server application capable of managing one or more sockets at a time.

Figure 8-6. WSAEventSelect *I/O model server sample code*

```
SOCKET Socket[WSA_MAXIMUM_WAIT_EVENTS];
WSAEVENT Event[WSA_MAXIMUM_WAIT_EVENTS];
SOCKET Accept, Listen;
DWORD EventTotal = 0;
DWORD Index;

// Set up a TCP socket for listening on port 5150
Listen = socket (PF_INET, SOCK_STREAM, 0);

InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(5150);

bind(Listen, (PSOCKADDR) &InternetAddr,
    Sizeof(InternetAddr));

NewEvent = WSACreateEvent();

WSAEventSelect(Listen, NewEvent,
    FD_ACCEPT | FD_CLOSE);

listen(Listen, 5);

Socket[EventTotal] = Listen;
Event[EventTotal] = NewEvent;
EventTotal++;

while(TRUE)
{
    // Wait for network events on all sockets
    Index = WSAWaitForMultipleEvents(EventTotal,
        EventArray, FALSE, WSA_INFINITE, FALSE);
```

```
    WSAEnumNetworkEvents(
        SocketArray[Index - WSA_WAIT_EVENT_0],
        EventArray[Index - WSA_WAIT_EVENT_0],
        &NetworkEvents);

    // Check for FD_ACCEPT messages
    if (NetworkEvents.lNetworkEvents & FD_ACCEPT)
    {
        if (NetworkEvents.iErrorCode[FD_ACCEPT_BIT] != 0)
        {
            printf("FD_ACCEPT failed with error %d\n",
                NetworkEvents.iErrorCode[FD_ACCEPT_BIT]);
            break;
        }

        // Accept a new connection, and add it to the
        // socket and event lists
        Accept = accept(
            SocketArray[Index - WSA_WAIT_EVENT_0],
            NULL, NULL);

        // We cannot process more than
        // WSA_MAXIMUM_WAIT_EVENTS sockets, so close
        // the accepted socket
        if (EventTotal > WSA_MAXIMUM_WAIT_EVENTS)
        {
            printf("Too many connections");
            closesocket(Accept);
            break;
        }

        NewEvent = WSACreateEvent();

        WSAEventSelect(Accept, NewEvent,
            FD_READ | FD_WRITE | FD_CLOSE);

        Event[EventTotal] = NewEvent;
        Socket[EventTotal] = Accept;
        EventTotal++;

        printf("Socket %d connected\n", Accept);
    }

    // Process FD_READ notification
    if (NetworkEvents.lNetworkEvents & FD_READ)
    {
        if (NetworkEvents.iErrorCode[FD_READ_BIT] != 0)
        {
            printf("FD_READ failed with error %d\n",
                NetworkEvents.iErrorCode[FD_READ_BIT]);
            break;
        }
```

```
        // Read data from the socket
        recv(Socket[Index - WSA_WAIT_EVENT_0],
            buffer, sizeof(buffer), 0);
    }

    // Process FD_WRITE notification
    if (NetworkEvents.lNetworkEvents & FD_WRITE)
    {
        if (NetworkEvents.iErrorCode[FD_WRITE_BIT] != 0)
        {
            printf("FD_WRITE failed with error %d\n",
                NetworkEvents.iErrorCode[FD_WRITE_BIT]);
            break;
        }

        send(Socket[Index - WSA_WAIT_EVENT_0],

            buffer, sizeof(buffer), 0);
    }

    if (NetworkEvents.lNetworkEvents & FD_CLOSE)
    {
        if (NetworkEvents.iErrorCode[FD_CLOSE_BIT] != 0)
        {
            printf("FD_CLOSE failed with error %d\n",
                NetworkEvents.iErrorCode[FD_CLOSE_BIT]);
            break;
        }

        closesocket(Socket[Index - WSA_WAIT_EVENT_0]);

        // Remove socket and associated event from
        // the Socket and Event arrays and decrement
        // EventTotal
        CompressArrays(Event, Socket, &EventTotal);
    }
}
```

The Overlapped Model

The overlapped I/O model in Winsock offers applications better system performance than any of the I/O models explained so far. The basic design of the overlapped model allows your application to post one or more Winsock I/O requests at a time using an overlapped data structure. At a later point, the application can service the submitted requests after they have completed. This model is available on all Windows platforms except Windows CE. The overall design of the model is based on the Win32 overlapped I/O mechanisms available for performing I/O operations on devices using the *ReadFile* and *WriteFile* functions.

Originally, the Winsock overlapped I/O model was available only to Winsock 1.1 applications running on Windows NT. Applications could take advantage of the model by calling *ReadFile* and *WriteFile* on a socket handle and specifying an overlapped structure that we will describe later. Since the release of Winsock 2, overlapped I/O has been incorporated into new Winsock functions, such as *WSASend* and *WSARecv*. As a result, the overlapped I/O model is now available on all Windows platforms that feature Winsock 2.

NOTE

With the release of Winsock 2, overlapped I/O can still be used with the functions *ReadFile* and *WriteFile* under Windows NT and Windows 2000. However, this functionality was not added to Windows 95 and Windows 98. For compatibility across platforms and for performance reasons, you should always consider using the *WSARecv* and *WSASend* functions instead of the Win32 *ReadFile* and *WriteFile* functions. This section will only describe how to use overlapped I/O through the new Winsock 2 functions.

To use the overlapped I/O model on a socket, you must first create a socket by using the flag *WSA_FLAG_OVERLAPPED*, as follows:

```
s = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED);
```

If you create a socket using the *socket* function instead of the *WSASocket* function, *WSA_FLAG_OVERLAPPED* is implied. After you successfully create a socket and bind it to a local interface, overlapped I/O operations can commence by calling the Winsock functions listed below and specifying an optional *WSAOVERLAPPED* structure.

- *WSASend*
- *WSASendTo*
- *WSARecv*
- *WSARecvFrom*
- *WSAIoctl*
- *AcceptEx*
- *TransmitFile*

As you probably already know, each one of these functions is associated with sending data, receiving data, and accepting connections on a socket. As a result, this activity can potentially take a long time to complete. This is why each function can accept a *WSAOVERLAPPED* structure as a parameter. When these functions are called with a *WSAOVERLAPPED* structure, they complete immediately—regardless of whether the socket is set to blocking mode (described at the beginning of this chapter). They rely on the *WSAOVERLAPPED* structure to manage the return of an I/O request. There are essentially two methods for managing the completion of an overlapped I/O request: your application can wait for *event object notification,* or it can process completed requests through *completion routines*. The functions listed above (except *AcceptEx*) have another parameter in common: *lpCompletionROUTINE*. This parameter is an optional pointer to a completion routine function that gets called when an overlapped request completes. We will explore the event notification method next. Later in this chapter, you will learn how to use optional completion routines instead of events to process completed overlapped requests.

Event notification

The event notification method of overlapped I/O requires associating Win32 event objects with *WSAOVERLAPPED* structures. When I/O calls such as *WSASend* and *WSARecv* are made using a *WSAOVERLAPPED* structure, they return immediately. Typically you will find that these I/O calls fail with the return value *SOCKET_ERROR.* The *WSAGetLastError* function reports a *WSA_IO_PENDING* error status. This error status simply means that the I/O operation is in progress. At some later time, your application will need to determine when an overlapped I/O request completes by waiting on the event object associated with the *WSAOVERLAPPED* structure. The *WSAOVERLAPPED* structure provides the communication medium between the initiation of an overlapped I/O request and its subsequent completion, and is defined as

```
typedef struct WSAOVERLAPPED
{
    DWORD    Internal;
    DWORD    InternalHigh;
    DWORD    Offset;
    DWORD    OffsetHigh;
    WSAEVENT hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

The *Internal, InternalHigh, Offset*, and *OffsetHigh* fields are all used internally by the system and should not be manipulated or used directly by an application. The *hEvent* field, on the other hand, is a special field that allows an application to associate an event object handle with a socket. You might be wondering how to get an event object handle to assign to this field. As we described in the *WSAEventSelect* model, you can use the *WSACreateEvent* function to create an event object handle. Once an event handle is created, simply assign the overlapped structure's *hEvent* field to the event handle and begin calling a Winsock function—such as *WSASend* or *WSARecv*—using the overlapped structure.

When an overlapped I/O request finally completes, your application is responsible for retrieving the overlapped results. In the event notification method, Winsock will change the event-signaling state of an event object that is associated with a *WSAOVERLAPPED* structure from nonsignaled to signaled when an overlapped request finally completes. Because an event object is assigned to the *WSAOVERLAPPED* structure, you can easily determine when an overlapped I/O call completes by calling the *WSAWaitForMultipleEvents* function, which we also described in the *WSAEventSelect* I/O model. *WSAWaitForMultipleEvents* waits a specified amount of time for one or more event objects to become signaled. We can't stress this point enough: remember that *WSAWaitForMultipleEvents* is capable of waiting on only 64 event objects at a time. Once you determine which overlapped request has completed, you need to determine the success or failure of the overlapped call by calling *WSAGetOverlappedResult*, which is defined as

```
BOOL WSAGetOverlappedResult(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags
);
```

The *s* parameter identifies the socket that was specified when the overlapped operation was started. The *lpOverlapped* parameter is a pointer to the *WSAOVERLAPPED* structure that was specified when the overlapped operation was started. The *lpcbTransfer* parameter is a pointer to a *DWORD* variable that receives the number of bytes that were actually transferred by an overlapped send or receive operation. The *fWait* parameter determines whether the function should wait for a pending overlapped operation to complete. If *fWait* is *TRUE*, the function does not return until the operation has been completed. If *fWait* is *FALSE* and the operation is still pending, *WSAGetOverlappedResult* returns *FALSE* with the error *WSA_IO_INCOMPLETE*. Since in our case we waited on a signaled event for overlapped completion, this parameter has no effect. The final parameter, *lpdwFlags,* is a pointer to a *DWORD* that will receive resulting flags if the originating overlapped call was made with the *WSARecv* or the *WSARecvFrom* function.

If the *WSAGetOverlappedResult* function succeeds, the return value is *TRUE*. This means that your overlapped operation has completed successfully and that the value pointed to by *lpcbTransfer* has been updated. If the return value is *FALSE*, one of the following statements is true:

- The overlapped I/O operation is still pending (as described above).
- The overlapped operation completed, but with errors.

- The overlapped operation's completion status could not be determined because of errors in one or more of the parameters supplied to *WSAGetOverlappedResult*.

Upon failure, the value pointed to by *lpcbTransfer* will not be updated, and your application should call the *WSAGetLastError* function to determine the cause of the failure.

Figure 8-7 demonstrates how to structure a simple server application that is capable of managing overlapped I/O on one socket, using the event notification described above. The application outlines the following programming steps:

1. Create a socket, and begin listening for a connection on a specified port.
2. Accept an inbound connection.
3. Create a *WSAOVERLAPPED* structure for the accepted socket, and assign an event object handle to the structure. Also assign the event object handle to an event array to be used later by the *WSAWaitForMultipleEvents* function.
4. Post an asynchronous *WSARecv* request on the socket by specifying the *WSAOVERLAPPED* structure as a parameter.

> NOTE
>
> This function will normally fail with *SOCKET_ERROR* error status *WSA_IO_PENDING*.

5. Call *WSAWaitForMultipleEvents* using the event array, and wait for the event associated with the overlapped call to become signaled.
6. After *WSAWaitForMultipleEvents* completes, reset the event object by using *WSAResetEvent* with the event array, and process the completed overlapped request.
7. Determine the return status of the overlapped call by using *WSAGetOverlappedResult*.
8. Post another overlapped *WSARecv* request on the socket.
9. Repeat steps 5-8.

This example can easily be expanded to handle more than one socket by moving the overlapped I/O processing portion of the code to a separate thread and allowing the main application thread to service additional connection requests.

Figure 8-7. *Simple overlapped example using events*

```
void main(void)
{
    WSABUF DataBuf;
    DWORD EventTotal = 0;
    WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
    WSAOVERLAPPED AcceptOverlapped;
    SOCKET ListenSocket, AcceptSocket;

    // Step 1:
    //  Start Winsock and set up a listening socket
    ...
```

```
    // Step 2:
    //  Accept an inbound connection
    AcceptSocket = accept(ListenSocket, NULL, NULL);

    // Step 3:
    //  Set up an overlapped structure

    EventArray[EventTotal] = WSACreateEvent();

    ZeroMemory(&AcceptOverlapped,
        sizeof(WSAOVERLAPPED));

    AcceptOverlapped.hEvent = EventArray[EventTotal];

    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = buffer;

    EventTotal++;

    // Step 4:
    //  Post a WSARecv request to begin receiving data
    //  on the socket

    WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes,
        &Flags, &AcceptOverlapped, NULL);

    // Process overlapped receives on the socket.

    while(TRUE)
    {
        // Step 5:
        //  Wait for the overlapped I/O call to complete
        Index = WSAWaitForMultipleEvents(EventTotal,
            EventArray, FALSE, WSA_INFINITE, FALSE);

        // Index should be 0 because we
        // have only one event handle in EventArray

        // Step 6:
        //  Reset the signaled event
        WSAResetEvent(
            EventArray[Index - WSA_WAIT_EVENT_0]);

        // Step 7:
        //  Determine the status of the overlapped
        //  request
        WSAGetOverlappedResult(AcceptSocket,
            &AcceptOverlapped, &BytesTransferred,
            FALSE, &Flags);

        // First check to see whether the peer has closed
        // the connection, and if so, close the
        // socket
```

```
        if (BytesTransferred == 0)
        {
            printf("Closing socket %d\n", AcceptSocket);

            closesocket(AcceptSocket);

            WSACloseEvent(
                EventArray[Index - WSA_WAIT_EVENT_0]);
            return;
        }

        // Do something with the received data.
        // DataBuf contains the received data.
        ...

        // Step 8:
        //  Post another WSARecv() request on the socket

        Flags = 0;
        ZeroMemory(&AcceptOverlapped,
            sizeof(WSAOVERLAPPED));

        AcceptOverlapped.hEvent = EventArray[Index -
            WSA_WAIT_EVENT_0];

        DataBuf.len = DATA_BUFSIZE;
        DataBuf.buf = Buffer;

        WSARecv(AcceptSocket, &DataBuf, 1,
            &RecvBytes, &Flags, &AcceptOverlapped,
            NULL);
    }
}
```

On Windows NT and Windows 2000, the overlapped I/O model also allows applications to accept connections in an overlapped fashion by calling the *AcceptEx* function on a listening socket. *AcceptEx* is a special Winsock 1.1 extension function that is available in the Mswsock.h header file and the Mswsock.lib library file. This function was originally intended to work with Win32 overlapped I/O on Windows NT and Windows 2000, but it also works with overlapped I/O in Winsock 2. *AcceptEx* is defined as

```
BOOL AcceptEx (
    SOCKET sListenSocket,
    SOCKET sAcceptSocket,
    PVOID lpOutputBuffer,
    DWORD dwReceiveDataLength,
    DWORD dwLocalAddressLength,
    DWORD dwRemoteAddressLength,
    LPDWORD lpdwBytesReceived,
    LPOVERLAPPED lpOverlapped
);
```

The *sListenSocket* parameter represents a listening socket. The *sAcceptSocket* parameter is a socket to accept an incoming connection. The *AcceptEx* function is different from the *accept* function in that you have to supply the accepted socket instead of having the function create it for you. Supplying the socket requires you to call the *socket* or *WSASocket* function to create a socket that you can pass to *AcceptEx* via the *sAcceptSocket* parameter. The *lpOutputBuffer* parameter is a special buffer because it receives three pieces of data: the local address of the server, the remote address of the client, and the first block of data sent on a new connection. The *dwReceiveDataLength* parameter specifies the number of bytes in *lpOutputBuffer* used for receiving data. If this parameter is specified as 0, no data will be received in conjunction with accepting the connection. The *dwLocalAddressLength* and *dwRemoteAddressLength* parameters represent how many bytes in *lpOutputBuffer* are reserved for storing local and remote address information when a socket is accepted. These buffer sizes must be at least 16 bytes more than the maximum address length for the transport protocol in use. For example, if you are using the TCP/IP protocol, the size should be set to the size of a *SOCKADDR_IN* structure + 16 bytes. The *lpdwBytesReceived* parameter returns the number of data bytes received. This parameter is set only if the operation completes synchronously. If the *AcceptEx* function returns *ERROR_IO_PENDING*, this parameter is never set and you must obtain the number of bytes read from the completion notification mechanism. The final parameter, *lpOverlapped*, is an *OVERLAPPED* structure that allows *AcceptEx* to be used in an asynchronous fashion. As we mentioned earlier, this function works with event object notification only in an overlapped application because it does not feature a completion routine parameter.

A Winsock extension function named *GetAcceptExSockaddrs* parses out the local and remote address elements from *lpOutputBuffer*. *GetAcceptExSockaddrs* is defined as

```
VOID GetAcceptExSockaddrs(
    PVOID lpOutputBuffer,
    DWORD dwReceiveDataLength,
    DWORD dwLocalAddressLength,
    DWORD dwRemoteAddressLength,
    LPSOCKADDR *LocalSockaddr,
    LPINT LocalSockaddrLength,
    LPSOCKADDR *RemoteSockaddr,
    LPINT RemoteSockaddrLength
);
```

The *lpOutputBuffer* parameter should be set to the *lpOutputBuffer* returned from *AcceptEx*. The *dwReceiveDataLength*, *dwLocalAddressLength*, and *dwRemoteAddressLength* parameters should be set to the same values as the *dwReceiveDataLength*, *dwLocalAddressLength*, and *dwRemoteAddressLength* parameters that were passed to *AcceptEx*. The *LocalSockaddr* and *RemoteSockaddr* parameters, which are pointers to *SOCKADDR* structures with the local and remote address information, receive a pointer offset from the originating *lpOutputBuffer* parameter. This makes it easy to reference the elements of a *SOCKADDR* structure from the address information contained in *lpOutputBuffer*. The *LocalSockaddrLength* and *RemoteSockaddrLength* parameters receive the size of the local and remote addresses.

Completion routines

Completion routines are the other method your application can use to manage completed overlapped I/O requests. Completion routines are simply functions that you optionally pass to an overlapped I/O request and that the system invokes when an overlapped I/O request completes. Their primary role is to service a completed I/O request using the caller's thread. Additionally, applications can continue overlapped I/O processing through the completion routine.

To use completion routines for overlapped I/O requests, your application must specify a completion routine, along with a *WSAOVERLAPPED* structure, to an I/O bound Winsock function (described earlier). A completion routine must have the following function prototype:

```
void CALLBACK CompletionROUTINE(
    DWORD dwError,
    DWORD cbTransferred,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwFlags
);
```

When an overlapped I/O request completes using a completion routine, the parameters contain the following information:

- The parameter *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*.
- The *cbTransferred* parameter specifies the number of bytes that were transferred during the overlapped operation.
- The *lpOverlapped* parameter is the *WSAOVERLAPPED* structure passed into the originating I/O call.
- The *dwFlags* parameter is not used and will be set to 0.

There is a major difference between overlapped requests submitted with a completion routine and overlapped requests submitted with an event object. The *WSAOVERLAPPED* structure's event field, *hEvent*, is not used, which means you cannot associate an event object with the overlapped request. Once you make an overlapped I/O call with a completion routine, your calling thread must eventually service the completion routine once it has completed. This requires you to place your calling thread in an *alertable wait state* and process the completion routine later, after the I/O operation has completed. The *WSAWaitForMultipleEvents* function can be used to put your thread in an alertable wait state. The catch is that you must also have at least one event object available for the *WSAWaitForMultipleEvents* function. If your application handles only overlapped requests with completion routines, you are not likely to have any event objects around for processing. As an alternative, your application can use the Win32 *SleepEx* function to set your thread in an alertable wait state. Of course, you can also create a dummy event object that is not associated with anything. If your calling thread is always busy and not in an alertable wait state, no posted completion routine will ever get called.

As you saw earlier, *WSAWaitForMultipleEvents* normally waits for event objects associated with *WSAOVERLAPPED* structures. This function is also designed to place your thread in an alertable wait state and to process completion routines for completed overlapped I/O requests if you set the parameter *fAlertable* to *TRUE*. When overlapped I/O requests complete with a completion routine, the return value is *WSA_IO_COMPLETION* instead of an event object index in the event array. The *SleepEx* function provides the same behavior as *WSAWaitForMultipleEvents* except that it does not need any event objects. The *SleepEx* function is defined as

```
DWORD SleepEx(
    DWORD dwMilliseconds,
    BOOL bAlertable
);
```

The *dwMilliseconds* parameter defines how long in milliseconds *SleepEx* will wait. If *dwMilliseconds* is set to *INFINITE*, *SleepEx* waits indefinitely. The *bAlertable* parameter determines how a completion routine will execute. If *bAlertable* is set to *FALSE* and an I/O completion callback occurs, the I/O completion function is not executed and the function does not return until the wait period specified in *dwMilliseconds* has elapsed. If it is set to *TRUE*, the completion routine executes and the *SleepEx* function returns *WAIT_IO_COMPLETION*.

Figure 8-8 outlines how to structure a simple server application that is capable of managing one socket request using completion routines as described above. The application illustrates the following programming steps:

1. Create a socket and begin listening for a connection on a specified port.
2. Accept an inbound connection.
3. Create a *WSAOVERLAPPED* structure for the accepted socket.
4. Post an asynchronous *WSARecv* request on the socket by specifying the *WSAOVERLAPPED* structure as a parameter and supplying a completion routine.
5. Call *WSAWaitForMultipleEvents* with the *fAlertable* parameter set to *TRUE*, and wait for an overlapped request to complete. When an overlapped request completes, the completion routine automatically executes and *WSAWaitForMultipleEvents* returns *WSA_IO_COMPLETION*. Inside the completion routine, post another overlapped *WSARecv* request with a completion routine.
6. Verify that *WSAWaitForMultipleEvents* returns *WSA_IO_COMPLETION*.
7. Repeat steps 5 and 6.

Figure 8-8. *Simple overlapped sample using completion routines*

```
SOCKET AcceptSocket;
WSABUF DataBuf;

void main(void)
{
    WSAOVERLAPPED Overlapped;

    // Step 1:
    //  Start Winsock, and set up a listening socket
    ...

    // Step 2:
    //  Accept a new connection
    AcceptSocket = accept(ListenSocket, NULL, NULL);

    // Step 3:
    //  Now that we have an accepted socket, start
    //  processing I/O using overlapped I/O with a
    //  completion routine. To get the overlapped I/O
    //  processing started, first submit an
    //  overlapped WSARecv() request.

    Flags = 0;

    ZeroMemory(&Overlapped, sizeof(WSAOVERLAPPED));

    DataBuf.len = DATA_BUFSIZE;
```

```
    DataBuf.buf = Buffer;

    // Step 4:
    //  Post an asynchronous WSARecv() request
    //  on the socket by specifying the WSAOVERLAPPED
    //  structure as a parameter, and supply
    //  the WorkerRoutine function below as the
    //  completion routine

    if (WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes,
        &Flags, &Overlapped, WorkerRoutine)
        == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSA_IO_PENDING)
        {
            printf("WSARecv() failed with error %d\n",
                WSAGetLastError());
            return;
        }
    }

    // Since the WSAWaitForMultipleEvents() API
    // requires waiting on one or more event objects,
    // we will have to create a dummy event object.
    // As an alternative, we can use SleepEx()
    // instead.

    EventArray[0] = WSACreateEvent();

    while(TRUE)
    {
        // Step 5:
        Index = WSAWaitForMultipleEvents(1, EventArray,
            FALSE, WSA_INFINITE, TRUE);

        // Step 6:
        if (Index == WAIT_IO_COMPLETION)
        {
            // An overlapped request completion routine
            // just completed. Continue servicing
            // more completion routines.
            break;
        }
        else
        {
            // A bad error occurred--stop processing!
            // If we were also processing an event
            // object, this could be an index to
            // the event array.

            return;
        }
    }
```

```
}

void CALLBACK WorkerRoutine(DWORD Error,
                            DWORD BytesTransferred,
                            LPWSAOVERLAPPED Overlapped,
                            DWORD InFlags)
{
    DWORD SendBytes, RecvBytes;
    DWORD Flags;

    if (Error != 0 || BytesTransferred == 0)
    {
        // Either a bad error occurred on the socket
        // or the socket was closed by a peer
        closesocket(AcceptSocket);
        return;
    }

    // At this point, an overlapped WSARecv() request
    // completed successfully. Now we can retrieve the
    // received data that is contained in the variable
    // DataBuf. After processing the received data, we
    // need to post another overlapped WSARecv() or
    // WSASend() request. For simplicity, we will post
    // another WSARecv() request.

    Flags = 0;

    ZeroMemory(&Overlapped, sizeof(WSAOVERLAPPED));

    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = Buffer;

    if (WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes,
        &Flags, &Overlapped, WorkerRoutine)
        == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSA_IO_PENDING )
        {
            printf("WSARecv() failed with error %d\n",
                WSAGetLastError());
            return;
        }
    }
}
```

The Completion Port Model

The completion port model is by far the most complicated I/O model. However, it offers the best system performance possible when an application has to manage many sockets at once. Unfortunately, it's available only on Windows NT and Windows 2000. Because of the complexity of its design, you should consider using the completion port model only if you need your application to manage hundreds or even thousands of sockets

simultaneously and you want your application to scale well when more CPUs are added to the system. The most important point to remember is that the I/O completion port model is your best choice if you are developing a high-performance server for Windows NT or Windows 2000 that is expected to service many socket I/O requests (a Web server, for example).

Essentially the completion port model requires you to create a Win32 completion port object that will manage overlapped I/O requests using a specified number of threads to service the completed overlapped I/O requests. Note that a completion port is actually a Win32, Windows NT, and Windows 2000 I/O construct that is capable of accepting more than just socket handles. However, this section will describe only how to take advantage of the completion port model by using socket handles. To begin using this model, you are required to create an I/O completion port object that will be used to manage multiple I/O requests for any number of socket handles. This is accomplished by calling the *CreateIoCompletionPort* function, which is defined as

```
HANDLE CreateIoCompletionPort(
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort,
    DWORD CompletionKey,
    DWORD NumberOfConcurrentThreads
);
```

Before examining the parameters in detail, be aware that this function is actually used for two distinct purposes:

- To create a completion port object
- To associate a handle with a completion port

When you initially create a completion port, the only parameter of interest is *NumberOfConcurrentThreads*; the first three parameters are ignored. The *NumberOfConcurrentThreads* parameter is special in that it defines the number of threads that are allowed to execute concurrently on a completion port. Ideally, you want only one thread per processor to service the completion port to avoid thread context switching. The value 0 for this parameter tells the system to allow as many threads as there are processors in the system. You can use the code below to create an I/O completion port.

```
CompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE,
    NULL, 0, 0);
```

This will return a handle that is used to identify the completion port when a socket handle is assigned to it.

Worker threads and completion ports

After a completion port is successfully created, you can begin to associate socket handles with the object. Before associating sockets, though, you have to create one or more worker threads to service the completion port when socket I/O requests are posted to the completion port object. At this point, you might wonder how many threads should be created to service the completion port. This is actually one of the more complicated aspects of the completion port model because the number needed to service I/O requests depends on the overall design of your application. It's important to note the distinction between number of concurrent threads to specify when calling *CreateIoCompletionPort* vs. the number of worker threads to create; they do not represent the same thing. We recommended earlier that you should have the *CreateIoCompletionPort* function specify one thread per processor to avoid thread context switching. The *NumberOfConcurrentThreads* parameter of *CreateIoCompletionPort* explicitly tells the system to allow only *n* threads to operate at a time on the completion port. If you create more than *n* worker threads on the completion port, only *n* threads will be allowed to operate at a time. (Actually, the system might exceed this value for a short amount of time, but the system will quickly bring it down to the value you specify in *CreateIoCompletionPort.*) You might be wondering why you would create more worker threads than the number specified by the *CreateIoCompletionPort* call. As we mentioned earlier, this depends on the overall design of your application. If one of your worker threads calls a function—such as *Sleep* or

WaitForSingleObject—and becomes suspended, another thread will be allowed to operate in its place. In other words, you always want to have as many threads available for execution as the number of threads you allow to execute in the *CreateIoCompletionPort* call. Thus, if you expect your worker thread to ever become blocked, it is reasonable to create more worker threads than the value specified in *CreateIoCompletionPort*'s *NumberOfConcurrentThreads* parameter.

Once you have enough worker threads to service I/O requests on the completion port, you can begin to associate socket handles with the completion port. This requires calling the *CreateIoCompletionPort* function on an existing completion port and supplying the first three parameters—*FileHandle*, *ExistingCompletionPort*, and *CompletionKey*—with socket information. The *FileHandle* parameter represents a socket handle to associate with the completion port. The *ExistingCompletionPort* parameter identifies the completion port. The *CompletionKey* parameter identifies *per-handle data* that you can associate with a particular socket handle. Applications are free to store any type of information associated with a socket by using this key. We call it per-handle data because it represents data associated with a socket handle. It is useful to store the socket handle using the key as a pointer to a data structure containing the socket handle and other socket-specific information. As we will see later in this chapter, the thread routines that service the completion port can retrieve socket-handle-specific information using this key.

Let's begin to construct a basic application framework from what we've learned so far. Figure 8-9 demonstrates how to start developing an echo server application using the completion port model. In this figure, we take the following preparation steps:

1. Create a completion port. The fourth parameter is left as 0, specifying that only one worker thread per processor will be allowed to execute at a time on the completion port.

2. Determine how many processors exist on the system.

3. Create worker threads to service completed I/O requests on the completion port using processor information in step 2. In the case of this simple example, we create one worker thread per processor because we do not expect our threads to ever get in a suspended condition in which there would not be enough threads to execute for each processor. When the *CreateThread* function is called, you must supply a worker routine that the thread executes upon creation. We will discuss the worker thread's responsibilities later in this section.

4. Prepare a listening socket to listen for connections on port 5150.

5. Accept inbound connections using the *accept* function.

6. Create a data structure to represent per-handle data and save the accepted socket handle in the structure.

7. Associate the new socket handle returned from *accept* with the completion port by calling *CreateIoCompletionPort*. Pass the per-handle data structure to *CreateIoCompletionPort* via the completion key parameter.

8. Start processing I/O on the accepted connection. Essentially, you want to post one or more asynchronous *WSARecv* or *WSASend* requests on the new socket using the overlapped I/O mechanism. When these I/O requests complete, a worker thread services the I/O requests and continues processing future I/O requests, as we will see later in the worker routine specified in step 3.

9. Repeat steps 5-8 until server terminates.

Figure 8-9. *Setting up a completion port*

```
StartWinsock();

// Step 1:
// Create an I/O completion port

CompletionPort = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE, NULL, 0, 0);

// Step 2:
```

```
// Determine how many processors are on the system

GetSystemInfo(&SystemInfo);

// Step 3:
// Create worker threads based on the number of
// processors available on the system. For this
// simple case, we create one worker thread for each
// processor.

for(i = 0; i < SystemInfo.dwNumberOfProcessors;
    i++)
{
    HANDLE ThreadHandle;

    // Create a server worker thread, and pass the
    // completion port to the thread. NOTE: the
    // ServerWorkerThread procedure is not defined
    // in this listing.

    ThreadHandle = CreateThread(NULL, 0,
        ServerWorkerThread, CompletionPort,
        0, &ThreadID);

    // Close the thread handle
    CloseHandle(ThreadHandle);
}

// Step 4:
// Create a listening socket

Listen = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED);

InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(5150);

bind(Listen, (PSOCKADDR) &InternetAddr,
    sizeof(InternetAddr));

// Prepare socket for listening

listen(Listen, 5);

while(TRUE)
{
    // Step 5:
    // Accept connections and assign to the completion
    // port

    Accept = WSAAccept(Listen, NULL, NULL, NULL, 0);
```

```
    // Step 6:
    // Create per-handle data information structure to
    // associate with the socket
    PerHandleData = (LPPER_HANDLE_DATA)
        GlobalAlloc(GPTR, sizeof(PER_HANDLE_DATA));

    printf("Socket number %d connected\n", Accept);
    PerHandleData->Socket = Accept;

    // Step 7:
    // Associate the accepted socket with the
    // completion port

    CreateIoCompletionPort((HANDLE) Accept,
        CompletionPort, (DWORD) PerHandleData, 0);

    // Step 8:
    //  Start processing I/O on the accepted socket.
    //  Post one or more WSASend() or WSARecv() calls
    //  on the socket using overlapped I/O.
    WSARecv(...);
}
```

Completion ports and overlapped I/O

After associating a socket handle with a completion port, you can begin processing I/O requests by posting send and receive requests on the socket handle. You can now start to rely on the completion port for I/O completion notification. Essentially, the completion port model takes advantage of the Win32 overlapped I/O mechanism in which Winsock API calls such as *WSASend* and *WSARecv* return immediately when called. It is up to your application to retrieve the results of the calls at a later time through an *OVERLAPPED* structure. In the completion port model, this is accomplished by having one or more worker threads wait on the completion port using the *GetQueuedCompletionStatus* function, which is defined as

```
BOOL GetQueuedCompletionStatus(
    HANDLE CompletionPort,
    LPDWORD lpNumberOfBytesTransferred,
    LPDWORD lpCompletionKey,
    LPOVERLAPPED * lpOverlapped,
    DWORD dwMilliseconds
);
```

The *CompletionPort* parameter represents the completion port to wait on. The *lpNumberOfBytesTransferred* parameter receives the number of bytes transferred after a completed I/O operation, such as *WSASend* or *WSARecv*. The *lpCompletionKey* parameter returns per-handle data for the socket that was originally passed into the *CreateIoCompletionPort* function. As we mentioned earlier, we recommend saving the socket handle in this key. The *lpOverlapped* parameter receives the overlapped result of the completed I/O operation. This is actually an important parameter because it can be used to retrieve *per I/O_operation data*. The final parameter, *dwMilliseconds*, specifies the number of milliseconds that the caller is willing to wait for a completion packet to appear on the completion port. If you specify *INFINITE*, the call waits forever.

Per-handle data and per – I/O operation data

When a worker thread receives I/O completion notification from the *GetQueuedCompletionStatus* API call, the *lpCompletionKey* and *lpOverlapped* parameters contain socket information that can be used to continue processing I/O on a socket through the completion port. Two types of important socket data are available through these parameters: per-handle data and per-I/O operation data.

The *lpCompletionKey* parameter contains what we call per-handle data because the data is related to a socket handle when a socket is first associated with the completion port. This is the data that is passed as the *CompletionKey* parameter of the *CreateIoCompletionPort* API call. As we noted earlier, your application can pass any type of socket information through this parameter. Typically, applications will store the socket handle related to the I/O request here.

The *lpOverlapped* parameter contains an *OVERLAPPED* structure followed by what we call per-I/O operation data, which is anything that your worker thread will need to know when processing a completion packet (echo the data back, accept the connection, post another read, and so on). Per-I/O operation data is any number of bytes attached to the end of an *OVERLAPPED* structure that you pass into a function that expects an *OVERLAPPED* structure. A simple way to make this work is to define a structure and place an *OVERLAPPED* structure as the first element of the new structure. For example, we declare the following data structure to manage per-I/O operation data:

```
typedef struct
{
    OVERLAPPED Overlapped;
    WSABUF     DataBuf;
    CHAR       Buffer[DATA_BUFSIZE];
    BOOL       OperationType;
} PER_IO_OPERATION_DATA;
```

This structure demonstrates some important data elements you might want to relate to an I/O operation, such as the type of I/O operation (a send or receive request) that just completed. In this structure, we consider the data buffer for the completed I/O operation to be useful. To call a Winsock API function that expects an *OVERLAPPED* structure, you can either cast your structure as an *OVERLAPPED* pointer or simply dereference the *OVERLAPPED* element of your structure. For example,

```
PER_IO_OPERATION_DATA PerIoData;

// You would call a function either as
    WSARecv(socket, ..., (OVERLAPPED *)&PerIoData);
// or as
    WSARecv(socket, ..., &(PerIoData.Overlapped));
```

Later in the worker thread, when *GetQueuedCompletionStatus* returns with an overlapped structure (and completion key), you can determine which operation was posted on this handle by dereferencing the *OperationType* member. (Just cast the returned overlapped structure to your *PER_IO_OPERATION_DATA* structure.) One of the biggest benefits of per-I/O operation data is that it allows you to manage multiple I/O operations (read/write, multiple reads, multiple writes, and so on) on the same handle. You might ask why you would want to post more than one I/O operation at a time on a socket. The answer is scalability. For example, if you have a multiple-processor machine with a worker thread using each processor, you could potentially have several processors sending and receiving data on a socket at the same time.

To complete the simple echo server sample from above, we need to supply a *ServerWorkerThread* function. Figure 8-10 outlines how to develop a worker thread routine that uses per-handle data and per-I/O operation data to service I/O requests.

Figure 8-10. *Completion port worker thread*

```
DWORD WINAPI ServerWorkerThread(
    LPVOID CompletionPortID)
{
    HANDLE CompletionPort = (HANDLE) CompletionPortID;
    DWORD BytesTransferred;
    LPOVERLAPPED Overlapped;
    LPPER_HANDLE_DATA PerHandleData;
    LPPER_IO_OPERATION_DATA PerIoData;
    DWORD SendBytes, RecvBytes;
    DWORD Flags;

    while(TRUE)
    {
        // Wait for I/O to complete on any socket
        // associated with the completion port

        GetQueuedCompletionStatus(CompletionPort,
            &BytesTransferred,(LPDWORD)&PerHandleData,
            (LPOVERLAPPED *) &PerIoData, INFINITE);

        // First check to see whether an error has occurred
        // on the socket; if so, close the
        // socket and clean up the per-handle data
        // and per-I/O operation data associated with
        // the socket

        if (BytesTransferred == 0 &&
            (PerIoData->OperationType == RECV_POSTED ||
             PerIoData->OperationType == SEND_POSTED))
        {
            // A zero BytesTransferred indicates that the
            // socket has been closed by the peer, so
            // you should close the socket. Note:
            // Per-handle data was used to reference the
            // socket associated with the I/O operation.

            closesocket(PerHandleData->Socket);

            GlobalFree(PerHandleData);
            GlobalFree(PerIoData);
            continue;
        }

        // Service the completed I/O request. You can
        // determine which I/O request has just
        // completed by looking at the OperationType
        // field contained in the per-I/O operation data.

        if (PerIoData->OperationType == RECV_POSTED)
        {
            // Do something with the received data
            // in PerIoData->Buffer
        }
```

```
        // Post another WSASend or WSARecv operation.
        // As an example, we will post another WSARecv()
        // I/O operation.

        Flags = 0;

        // Set up the per-I/O operation data for the next
        // overlapped call
        ZeroMemory(&(PerIoData->Overlapped),
            sizeof(OVERLAPPED));

        PerIoData->DataBuf.len = DATA_BUFSIZE;
        PerIoData->DataBuf.buf = PerIoData->Buffer;
        PerIoData->OperationType = RECV_POSTED;

        WSARecv(PerHandleData->Socket,
            &(PerIoData->DataBuf), 1, &RecvBytes,
            &Flags, &(PerIoData->Overlapped), NULL);
    }
}
```

One final detail not outlined in the simple server examples in Figures 8-9 and 8-10 or on the companion CD is how to properly close an I/O completion port, especially if you have one or more threads in progress performing I/O on several sockets. The main thing to avoid is freeing an *OVERLAPPED* structure when an overlapped I/O operation is in progress. The best way to prevent this is to call *closesocket* on every socket handle—any overlapped I/O operations pending will complete. Once all socket handles are closed, you need to terminate all worker threads on the completion port. This can be accomplished by sending a special completion packet to each worker thread using the *PostQueuedCompletionStatus* function, which informs each thread to exit immediately. *PostQueuedCompletionStatus* is defined as

```
BOOL PostQueuedCompletionStatus(
    HANDLE CompletionPort,
    DWORD dwNumberOfBytesTransferred,
    DWORD dwCompletionKey,
    LPOVERLAPPED lpOverlapped
);
```

The *CompletionPort* parameter represents the completion port object to which you want to send a completion packet. The *dwNumberOfBytesTransferred*, *dwCompletionKey,* and *lpOverlapped* parameters each allow you to specify a value that will be sent directly to the corresponding parameter of the *GetQueuedCompletionStatus* function. Thus, when a worker thread receives the three passed parameters of *GetQueuedCompletionStatus*, it can determine when it should exit based on a special value set in one of the three parameters. For example, you could pass the value 0 in the *dwCompletionKey* parameter, which a worker thread could interpret as an instruction to terminate. Once all the worker threads are closed, you can close the completion port using the *CloseHandle* function and finally exit your program safely.

Other issues

Several techniques can further improve overall I/O performance of a socket application using completion ports. One technique worth considering is experimenting with socket buffer sizes to increase I/O performance and application scalability. For example, if your application uses one large buffer with only one *WSARecv* request instead of three small buffers with three *WSARecv* requests, your application will not scale well to multiprocessor

machines. This is because a single buffer can be processed by only one thread at a time. Furthermore, the single-buffer approach has performance consequences: you might not be keeping the network protocol driver busy enough if you are doing only one receive operation at a time. That is, if you wait for one *WSARecv* to complete before you receive more data, you effectively let the protocol rest between the *WSARecv* completion and the next receive.

Another performance gain worth considering results from using the socket options *SO_SNDBUF* and *SO_RCVBUF* to control the size of internal socket buffers. These options allow an application to change the size of the internal data buffer of a socket. If you set this value to 0, Winsock will use your application buffer directly in an overlapped I/O call to transmit data to and from the protocol stack, thereby reducing a buffer copy between your application and Winsock. The following code fragment demonstrates how to call the *setsockopt* function using the *SO_SNDBUF* option:

```
int nZero = 0;

setsockopt(socket, SOL_SOCKET, SO_SNDBUF,
    (char *)&nZero, sizeof(nZero));
```

Note that setting these buffer sizes to 0 has only a positive impact on your application when multiple I/O requests are posted at a given time. Chapter 9 describes these socket options in greater detail.

A final performance gain worth considering results from using the *AcceptEx* API call for connection requests that deliver small amounts of data. This allows your application to service an accept request and retrieve data through a single API call, thereby reducing the overhead of separate *accept* and *WSARecv* calls. As an added benefit, you can service *AcceptEx* requests using the completion port since it also features an *OVERLAPPED* structure. *AcceptEx* is useful if your server expects to handle a small amount of *recv-send* transactions once a connection is established (as with a Web server). Otherwise, if your application is performing hundreds or thousands of data transfers after accepting a connection, this operation offers no real gain.

On a final note, Winsock applications should not use the *ReadFile* and *WriteFile* Win32 functions for processing I/O on a completion port in Winsock. These functions do feature an *OVERLAPPED* structure and can be successfully used on a completion port; however, the *WSARecv* and *WSASend* functions are better optimized for processing I/O in Winsock 2. Using *ReadFile* and *WriteFile* involves making many more unnecessary kernel/user mode procedure call transitions, thread context switches, and parameter marshaling, resulting in a significant performance penalty.

I/O Model Consideration

By now you might be wondering how to choose the I/O model you should use when designing your application. As we've mentioned, each model has its strengths and weaknesses. All of the I/O models do require fairly complex programming when compared with developing a simple blocking-mode application with many servicing threads. We offer the suggestions below for client and server development.

Client development

When you are developing a client application that manages one or more sockets, we recommend using overlapped I/O or *WSAEventSelect* over the other I/O models for performance reasons. However, if you are developing a Windows-based application that manages window messages, the *WSAAsyncSelect* model might be a better choice because *WSAAsyncSelect* lends itself to the Windows message model, and your application is already set up for handling messages.

Server development

When you are developing a server that processes several sockets at a given time, we recommend using overlapped I/O over the other I/O models for performance reasons. However, if you expect your server to service a large number of I/O requests at any given time, you should consider using the I/O completion port model for even better performance.

Conclusion

At this point, we have covered the various I/O models available in Winsock. These models allow applications to tailor Winsock I/O according to their needs, from simple blocking I/O to high-performance completion port I/O for the maximum throughput possible. This chapter also concludes our discussion of the general-purpose aspect of Winsock. Up to this point, you have learned about available transport protocols, socket creation attributes, creating a basic client/server application, and other fundamental Winsock topics. In Chapters 9 through 14, we'll introduce specialized Winsock topics. The next chapter deals with socket options and ioctl commands that affect the behavior of both sockets and the underlying protocol.

Chapter 9

Socket Options and Ioctls

Once a socket has been created, various attributes can be manipulated via socket options and ioctl commands to affect the behavior of the socket. Some of these options simply return information, while others affect the behavior of the socket in your application. An ioctl is an I/O control command that also affects the behavior of the socket. This chapter is dedicated to discussing four Winsock functions: *getsockopt*, *setsockopt*, *ioctlsocket*, and *WSAIoctl*. Each function has numerous commands, many of which have never been properly documented. In the following sections, we will discuss the required parameters and available options for each function as well as the platforms that support those options. Every option is assumed to work on all Win32 platforms (Windows CE, Windows 95, Windows 98, Windows NT, and Windows 2000) unless otherwise noted. The only exception occurs when an option requires Winsock 2. Because Winsock 2 is not available on every platform, Winsock 2 ioctl commands and options are not supported on Windows CE or Windows 95 (unless the Winsock 2 update has been applied to Windows 95). Furthermore, recall that Windows CE does not support any protocol-specific options not pertaining to TCP/IP.

Most of these ioctl commands and options are defined in either Winsock.h or Winsock2.h, depending on whether they are specific to Winsock 1 or Winsock 2; however, a few of the options are specific either to the Microsoft provider or to a particular transport protocol. Microsoft-specific extensions are defined in both Winsock.h and Mswsock.h. Transport provider extensions are defined in their protocol-specific header files. For the transport-specific options, we will indicate the correct header file along with the option. Applications using the Microsoft-specific extensions must link with Mswsock.lib.

Socket Options

The *getsockopt* function is most frequently used to obtain information regarding the given socket. The prototype for this function is as follows:

```
int getsockopt (
    SOCKET s,
    int level,
    int optname,
    char FAR* optval,
    int FAR* optlen
);
```

The first parameter, *s*, is the socket on which you want to perform the specified option. This must be a valid socket for the given protocol you are using. A number of options are specific to a particular protocol and socket type, while others pertain to all types of sockets. This ties in with the second parameter, *level*. An option of level *SOL_SOCKET* means it is a generic option that isn't necessarily specific to a given protocol. We say "necessarily" because not all protocols implement each socket option of level *SOL_SOCKET*. For example, *SO_BROADCAST* puts the socket into broadcast mode, but not all supported protocols support the notion of broadcast sockets. The *optname* parameter is the actual option you are interested in. These option names are constant values defined in the Winsock header files. The most common and protocol-independent options (such as those with the *SOL_SOCKET* level) are defined in Winsock.h and Winsock2.h. Each specific protocol has its own header file that defines options specific to it. Finally, the *optval* and *optlen* parameters are the variables returned with the value of the desired option. In most cases—but not all—the option value is an integer.

The *setsockopt* function is used to set socket options on either a socket level or a protocol-specific level. The function is defined as

```
int setsockopt (
    SOCKET s,
    int level,
    int optname,
    const char FAR * optval,
    int optlen
);
```

The parameters are the same as in *getsockopt* except that you pass in a value as the *optval* and *optlen* parameters, which are the values to set for the specified option. As with *getsockopt*, *optval* is often, but not always, an integer. Consult each option for the specifics on what is passed as the option value.

The most common mistake associated with calling either *getsockopt* or *setsockopt* is attempting to obtain socket information for a socket whose underlying protocol doesn't possess that particular characteristic. For example, a socket of type *SOCK_STREAM* is not capable of broadcasting data; therefore, attempting to set or get the *SO_BROADCAST* option results in the error *WSAENOPROTOOPT*.

SOL_SOCKET Option Level

This section describes the socket options that return information based on the characteristics of the socket itself and are not specific to the protocol of that socket.

SO_ACCEPTCONN

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Get only	1+	If *TRUE*, socket is in listening mode.

If the socket has been put into listening mode by the *listen* function, this option returns *TRUE*. Sockets of type *SOCK_DGRAM* do not support this option.

SO_BROADCAST

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, socket is configured for sending broadcast messages.

If the given socket has been configured for sending or receiving broadcast data, querying this socket option returns *TRUE*. Use *setsockopt* with *SO_BROADCAST* to enable broadcast abilities on the socket. This option is valid for sockets that aren't of type *SOCK_STREAM*.

Broadcasting is the ability to send data so that every machine on the local subnet receives the data. Of course, there must be some process on each machine that listens for incoming broadcast data. The drawback of broadcasting is that if many processes are all sending broadcast data, the network can become saturated and network performance suffers. In order to receive a broadcast message, you must enable the broadcast option and then use one of the datagram receive functions, such as *recvfrom* or *WSARecvfrom*. You can also connect the socket to the broadcast address by calling *connect* or *WSAConnect* and then use *recv* or *WSARecv*. For UDP broadcasts, you must specify a port number to send the datagram to; likewise, the receiver must request to receive the broadcast data on that port also. The following code example illustrates how to send a broadcast message with UDP:

```
SOCKET       s;
BOOL         bBroadcast;
char         *sMsg = "This is a test";
SOCKADDR_IN bcast;

s = WSASocket(AF_INET, SOCK_DGRAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
bBroadcast = TRUE;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, (char *)&bBroadcast,
    sizeof(BOOL));
bcast.sin_family = AF_INET;
bcast.sin_addr.s_addr = inet_addr(INADDR_BROADCAST);
bcast.sin_port = htons(5150);
sendto(s, sMsg, strlen(sMsg), 0, (SOCKADDR *)&bcast, sizeof(bcast));
```

For UDP, a special broadcast address exists to which broadcast data should be sent. This address is 255.255.255.255. A #*define* directive for *INADDR_BROADCAST* is provided to make things a bit simpler and easier to read.

AppleTalk is another protocol capable of sending broadcast messages. AppleTalk also has a special address used by broadcast data. You learned in Chapter 6 that an AppleTalk address has three parts: network, node, and socket (destination). For broadcasting, set the destination to *ATADDR_BROADCAST* (0xFF), which causes the datagram to be sent to all endpoints on the given network.

Normally, you need to set only the *SO_BROADCAST* option when sending broadcast datagrams. To receive a broadcast datagram, you need to be listening only for incoming datagrams on that specified port. However, on

Windows 95 when using IPX, the receiving socket must set the *SO_BROADCAST* option in order to receive broadcast data, as described in Knowledge Base article Q137914, which can be found at http://support.microsoft.com/support/search. This is a bug in Windows 95.

SO_CONNECT_TIME

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Get only	1+	Returns the number of seconds the socket has been connected

SO_CONNECT_TIME is a Microsoft-specific option that returns the number of seconds a connection has been established. The most frequent use of this option is with the *AcceptEx* function. *AcceptEx* requires that a valid socket handle be passed for the incoming client connection. This option can be called on the client's *SOCKET* handle to determine whether the connection has been made and how long the connection as been established. If the socket is not currently connected, the value returned is 0xFFFFFFFF.

SO_DEBUG

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, debug output is enabled.

Winsock service providers are encouraged (but not required) to supply output debug information if the *SO_DEBUG* option is set by an application. How the debug information is presented depends on the underlying service provider's implementation. To turn debug information on, call *setsockopt* with *SO_DEBUG* and a Boolean variable set to *TRUE*. Calling *getsockopt* with *SO_DEBUG* returns *TRUE* or *FALSE* if debugging is enabled or disabled, respectively. Unfortunately, no Win32 platform currently implements the *SO_DEBUG* option, as described in Knowledge Base article Q138965. No error is returned when the option is set, but the underlying network provider ignores the option.

SO_DONTLINGER

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, *SO_LINGER* is disabled.

For protocols that support graceful socket connection closure, a mechanism is implemented so that if one or both sides close the socket, any data still pending or in transmission will be sent or received by both parties. It is possible, with *setsockopt* and the *SO_LINGER* option, to change this behavior so that after a specified period of time, the socket and all its resources will be torn down. Any pending or arriving data associated with that socket is discarded and the peer's connection reset (*WSAECONNRESET*). The *SO_DONTLINGER* option can be checked to ensure that a linger period has not been set. Calling *getsockopt* with *SO_DONTLINGER* will return a Boolean *TRUE* or *FALSE* if a linger value is set or not set, respectively. A call to *setsockopt* with *SO_DONTLINGER* disables lingering. Sockets of type *SOCK_DGRAM* do not support this option.

SO_DONTROUTE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, messages are sent directly to the network interface without consulting the routing table.

The *SO_DONTROUTE* option tells the underlying network stack to ignore the routing table and to send the data out

on the interface the socket is bound to. For example, if you create a UDP socket and bind it to interface A and then send a packet destined for a machine on the network attached to interface B, the packet will in fact be routed so that it is sent on interface B. Using *setsockopt* with the Boolean value *TRUE* prevents this because the packet goes out on the bound interface. The *getsockopt* function can be called to determine whether routing is enabled (which it is by default).

Calling this option on a Win32 platform will succeed; however, the Microsoft provider silently ignores the request and always uses the routing table to determine the appropriate interface for outgoing data.

SO_ERROR

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Get only	1+	Returns the error status

The *SO_ERROR* option returns and resets the per-socket-based error code, which is different from the per-thread-based error code that is handled using the *WSAGetLastError* and *WSASetLastError* function calls. A successful call using the socket does not reset the per-socket-based error code returned by the *SO_ERROR* option. Calling this option will not fail; however, the error value is not always updated immediately, so there is a possibility of this option returning 0 (indicating no error). It is best to use *WSAGetLastError* unless it is absolutely necessary to rely on the individual error code.

SO_EXCLUSIVEADDRUSE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	2+	If *TRUE*, the local port that the socket is bound to cannot be reused by another process.

This option is the complement of *SO_REUSEADDR*, which we will describe shortly. This option exists to prevent other processes from using the *SO_REUSEADDR* on a local address that your application is using. If two separate processes are bound to the same local address (assuming that *SO_REUSEADDR* is set earlier), which of the two sockets receives notifications for incoming connections is not defined. This is extremely unfortunate if your application is mission-critical. The *SO_EXCLUSIVEADDRUSE* option locks down the local address to which the socket is bound, so if any other process tries to use *SO_REUSEADDR* with the same local address, that process fails. Administrator rights are required to set this option. It is available only on Windows 2000.

SO_KEEPALIVE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, socket is configured to send keepalive messages on the session.

For a TCP-based socket, an application can request that the underlying service provider enable the use of keepalive packets on TCP connections by turning on the *SO_KEEPALIVE* socket option. On Win32 platforms, keepalives are implemented in accordance with section 4.2.3.6 of RFC 1122. If a connection is dropped as the result of keepalives, the error code *WSAENETRESET* is returned to any calls in progress on the socket, and any subsequent calls will fail with *WSAENOTCONN*. For the exact implementation details, consult the RFC. The important thing to note here is that keepalives are sent at intervals no less than 2 hours apart. The 2-hour keepalive time is configurable via the Registry; however, changing the default value changes the keepalive behavior for all TCP connections on the system, which is generally discouraged. Another solution is to implement your own keepalive strategy. Chapter 7 discusses this kind of strategy. Sockets of type *SOCK_DGRAM* do not support this option.

The Registry keys for keepalives are *KeepAliveInterval* and *KeepAliveTime*. Both keys store values of type *REG_DWORD* in milliseconds. The former key is the interval separating keepalive retransmissions until a response

is received; the latter entry controls how often TCP sends a keepalive packet in an attempt to verify that an ideal connection is still valid. In Windows 95 and Windows 98, these keys are located under the following Registry path:

```
\HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VxD\MSTCP
```

In Windows NT and Windows 2000, store the keys under

```
\HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\TCPIP\Parameters
```

With Windows 2000, a new socket ioctl command—*SIO_KEEPALIVE_VALS*—allows you to change the keepalive value and interval on a per-socket basis, as opposed to a system-wide basis. This ioctl command is described later in this chapter.

SO_LINGER

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
struct linger	Both	1+	Sets or gets the current linger values

SO_LINGER controls the action taken when unsent data is queued on a socket and a *closesocket* is performed. A call to *getsockopt* with this socket option returns the current linger times in a *linger* structure, which is defined as

```
struct linger {
    u_short l_onoff;
    u_short l_linger;
}
```

A nonzero value for *l_onoff* means that lingering is enabled, while *l_linger* is the timeout in seconds, at which point any pending data to be sent or received is discarded and the connection with the peer is reset. Conversely, you can call *setsockopt* to turn lingering on and specify the length of time before discarding any queued data. This is accomplished by setting the desired values in a variable of type *struct linger*. When setting a linger value with *setsockopt,* you must set the *l_onoff* field of the structure to a nonzero value. To turn lingering off once it has been enabled, you can call *setsockopt* with the *SO_LINGER* option and the *l_onoff* field of the *linger* structure set to 0, or you can call *setsockopt* with the *SO_DONTLINGER* option, passing the value *TRUE* for the *optval* parameter. Sockets of type *SOCK_DGRAM* do not support this option.

Setting the linger option directly affects how a connection behaves when the *closesocket* function is called. Table 9-1 lists these behaviors.

Table 9-1. *Linger options*

Option	*Interval*	*Type of Close*	*Wait for Close?*
SO_DONTLINGER	Not applicable	Graceful	No
SO_LINGER	0	Hard	No
SO_LINGER	Nonzero	Graceful	Yes

If *SO_LINGER* is set with a zero timeout interval (that is, the *linger* structure member *l_onoff* is not 0 and *l_linger* is 0), *closesocket* is not blocked, even if queued data has not yet been sent or acknowledged. This is called a hard,

or abortive, close because the socket's virtual circuit is reset immediately and any unsent data is lost. Any receive call on the remote side of the circuit fails with *WSAECONNRESET*.

If *SO_LINGER* is set with a nonzero timeout interval on a blocking socket, the *closesocket* call blocks on a blocking socket until the remaining data has been sent or until the timeout expires. This is called a graceful disconnect. If the timeout expires before all data has been sent, the Windows Sockets implementation terminates the connection before *closesocket* returns.

SO_MAX_MSG_SIZE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
unsigned int	Get only	2+	The maximum size of a message for a message-oriented socket

This is a get-only socket option that indicates the maximum outbound (send) size of a message for message-oriented socket types as implemented by a particular service provider. It has no meaning for byte-stream-oriented sockets. There is no provision for finding the maximum inbound message size.

SO_OOBINLINE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, any out-of-band data is returned in the normal data stream.

By default, out-of-band (OOB) data is not inlined. That means a call to a receive function (with the appropriate *MSG_OOB* flag set) returns the OOB data in a single call. If this option is set, the OOB data appears within the data stream returned from a receive call, and a call to *ioctlsocket* with the *SIOCATMARK* option is required to determine which byte is the OOB data. Sockets of type *SOCK_DGRAM* do not support this option. Unfortunately, this socket option is broken on all current Win32 implementations. See Chapter 7 for more details on OOB data.

SO_PROTOCOL_INFO

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSAPROTOCOL_INFO	Get only	2+	The protocol characteristics for the protocol that the socket is bound to

This is another get-only option that fills in the supplied *WSAPROTOCOL_INFO* structure with the characteristics of the protocol associated with the socket. See Chapter 6 for a description of the *WSAPROTOCOL_INFO* structure and its member fields.

SO_RCVBUF

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	Gets or sets the per-socket buffer size for receive operations

This is a simple option that either returns the size or sets the size of the buffer allocated to this socket for receiving data. When a socket is created, a send buffer and a receive buffer are assigned to the socket for sending and receiving data. When requesting to set the receive buffer size to a value, the call to *setsockopt* can succeed

even when the implementation does not provide the entire amount requested. To ensure that the requested buffer size is allocated, call *getsockopt* to get the actual size allocated. All Win32 platforms can get or set the receive buffer size except Windows CE, which does not allow you to change the value—you can get only the receive buffer size.

One possible reason for changing the buffer size is to specifically tailor buffer sizes according to your application's behavior. For example, when writing code to receive UDP datagrams, you should generally make the receive buffer size an even multiple of the datagram size. For overlapped I/O, setting the buffer sizes to 0 can increase performance in certain situations: when these buffers are nonzero, an extra memory copy is involved in moving data from the system buffer to the user-supplied buffer. If there is no intermediate buffer, data is immediately copied to the user-supplied buffer. The one caveat is that this is efficient only with multiple outstanding receive calls. Posting only a single receive can hurt performance, as the local system cannot accept any incoming data unless you have a buffer posted and ready to receive the data. For more information, see the section called "Other Issues" under the completion port I/O model in Chapter 8.

SO_REUSEADDR

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, the socket can be bound to an address already in use by another socket or to an address in the *TIME_WAIT* state.

By default, a socket cannot be bound to a local address that is already in use; however, occasionally it is necessary to reuse an address in this way. Remember from Chapter 7 that each connection is uniquely identified by the combination of its local and remote addresses. As long as the address to which you are connecting is unique in the slightest respect (such as a different port number in TCP/IP), the binding will be allowed.

The only exception is for a listening socket. Two separate sockets cannot bind to the same local interface (and port, in the case of TCP/IP) to await incoming connections. If two sockets are actively listening on the same port, the behavior is undefined as to which socket will receive notification of an incoming connection. The *SO_REUSEADDR* option is most useful in TCP when a server shuts down or exits abnormally so that the local address and port are in the *TIME_WAIT* state, which prevents any other sockets from binding to that port. By setting this option, the server can listen on the same local interface and port when it is restarted.

SO_SNDBUF

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	*TRUE* (nonzero) means socket is configured for sending broadcast messages.

This is a simple option that either returns the size or sets the size of the buffer allocated to this socket for sending data. When a socket is created, a send buffer and a receive buffer are assigned to the socket for sending and receiving data. When requesting to set the size of the send buffer, the call to *setsockopt* can succeed even when the implementation does not provide the entire amount requested. To ensure that the requested buffer size is allocated, call *getsockopt* to get the actual size allocated. All Win32 platforms can get or set the send buffer size except Windows CE, which does not allow you to change the value—you can get only the receive buffer size.

As with *SO_RCVBUF*, you can use the *SO_SNDBUF* option to set the size of the send buffer to 0. The advantage of the buffer size being 0 for blocking send calls is that when the call completes you know that your data is on the wire. Also, as in the case of a receive operation with a zero-length buffer, there is no extra memory copy of your data to system buffers. The drawback is that you lose the pipelining gained by the default stack buffering when the send buffers are nonzero in size. In other words, if you have a loop performing sends, the local network stack can copy your data to a system buffer to be sent when possible (depending on the I/O model being used). On the other hand, if your application is concerned with other logistics, disabling the send buffers can save you a few machine instructions in the memory copy. For additional information, see the section entitled "Other Issues" under the completion port I/O model in Chapter 8.

SO_TYPE

optval Type	Get/Set	Winsock Version	Description
int	Get only	1+	Returns the socket type (e.g., *SOCK_DGRAM*, *SOCK_STREAM*, etc.) of the given socket

The *SO_TYPE* option is a get-only option that simply returns the socket type of the given socket. The possible socket types are *SOCK_DGRAM*, *SOCK_STREAM*, *SOCK_SEQPACKET*, *SOCK_RDM*, and *SOCK_RAW*.

SO_SNDTIMEO

optval Type	Get/Set	Winsock Version	Description
int	Both	1+	Gets/sets the timeout value (in milliseconds) associated with sending data on the socket

The *SO_SNDTIMEO* option sets the timeout value on a blocking socket when calling a Winsock send function. The timeout value is an integer in milliseconds that indicates how long the send function should block when attempting to send data. If you need to use the *SO_SNDTIMEO* option and you use the *WSASocket* function to create the socket, you must specify *WSA_FLAG_OVERLAPPED* as part of *WSASocket*'s *dwFlags* parameter. Subsequent calls to any Winsock send function (*send*, *sendto*, *WSASend*, *WSASendTo*, and so on) block only for the amount of time specified. If the send operation cannot complete within that time, the call fails with error 10060 (*WSAETIMEDOUT*).

For performance reasons, this option was disabled in Windows CE 2.1. If you attempt to set this option, the option is silently ignored and no failure is returned. Previous versions of Windows CE do implement this option.

SO_RCVTIMEO

optval Type	Get/Set	Winsock Version	Description
int	Both	1+	Gets/sets the timeout value (in milliseconds) associated with receiving data on the socket

The *SO_RCVTIMEO* option sets the receive timeout value on a blocking socket. The timeout value is an integer in milliseconds that indicates how long a Winsock receive function should block when attempting to receive data. If you need to use the *SO_RCVTIMEO* option and you use the *WSASocket* function to create the socket, you must specify *WSA_FLAG_OVERLAPPED* as part of *WSASocket*'s *dwFlags* parameter. Subsequent calls to any Winsock receive function (*recv*, *recvfrom*, *WSARecv*, *WSARecvFrom*, and so on) block only for the amount of time specified. If no data arrives within that time, the call fails with the error 10060 (*WSAETIMEDOUT*).

For performance reasons, this option was disabled in Windows CE 2.1. If you attempt to set this option, it is silently ignored and no failure returns. Previous versions of Windows CE do implement this option.

SO_UPDATE_ACCEPT_CONTEXT

optval Type	Get/Set	Winsock Version	Description
SOCKET	Both	1+	Gets/sets the timeout value (in milliseconds) associated with receiving data on the socket

This option is a Microsoft-specific extension most commonly used in conjunction with the *AcceptEx* function. The unique characteristic of this function is that it is part of the Winsock 1 specification and allows the use of overlapped I/O for an accept call. The function takes the listening socket as a parameter as well as a socket

handle that becomes the accepted client. This socket option must be set in order for the characteristics of the listening socket to be carried over to the client socket. This is particularly important for Quality of Service (QOS)-enabled listening sockets. In order for the client socket to be QOS-enabled, this option must be set. To set this option on a socket, use the listening socket as the *SOCKET* parameter to *setsockopt* and pass the accepting socket handle (for example, the client handle) as *optval*. This option is specific to Windows NT and Windows 2000.

SOL_APPLETALK Option Level

The following options are socket options specific to the AppleTalk protocol and can be used only with sockets created using *socket* or *WSASocket* with the *AF_APPLETALK* flag. A majority of the options listed here deal with either setting or obtaining AppleTalk names. For more information on the AppleTalk address family, refer back to Chapter 6. Some AppleTalk socket options—such as *SO_DEREGISTER_NAME*—have more than one option name. In such cases, all the option's names can be used interchangeably.

SO_CONFIRM_NAME

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_NBP_TUPLE	Get only	1	Confirms that the given AppleTalk name is bound to the given address

The *SO_CONFIRM_NAME* option is used to verify that a given AppleTalk name is bound to the supplied address. This results in a Name Binding Protocol (NBP) lookup request being sent to the address to verify the name. If the call fails with the error *WSAEADDRNOTAVAIL*, the name is no longer bound to the address given.

SO_DEREGISTER_NAME, *SO_REMOVE_NAME*

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_REGISTER_NAME	Set only	1	Deregisters the given name from the network

This option is used to deregister a name from the network. If the name does not currently exist on the network, the call will return indicating success. Refer to the section entitled "Registering an AppleTalk Name" in Chapter 6 for a description of the *WSH_REGISTER_NAME* structure, which is simply another name for the *WSH_NBP_NAME* structure.

SO_LOOKUP_MYZONE, *SO_GETMYZONE*

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
char *	Get only	1	Returns the default zone on the network

This option returns the default zone on the network. The *optval* parameter to *getsockopt* should be a character string of at least 33 characters. Remember that the maximum length of an NBP name is *MAX_ENTITY_LEN*, which is defined as 32. The extra character is required for the null terminator.

SO_LOOKUP_NAME

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_LOOKUP_NAME	Get only	1	Looks up a specified NBP name and returns the matching tuples of names and NBP information

This option is used to look up a specified name on the network (for example, when a client wants to connect to a server). The well-known textual name must be resolved to an AppleTalk address before a connection can be established. See the section "Resolving an AppleTalk Name" in Chapter 6 for sample code on how to look up an AppleTalk name.

One thing to be aware of is that upon successful return, the *WSH_NBP_TUPLE* structures occupy the space in the supplied buffer after the *WSH_LOOKUP_NAME* information. That is, you should supply *getsockopt* with a buffer large enough to hold the *WSH_LOOKUP_NAME* information at the start of the buffer and a number of *WSH_NBP_TUPLE* structures in the remaining space. Figure 9-1 illustrates how the buffer should be prepared prior to the call (with respect to *WSH_LOOKUP_NAME*) and where the *WSH_NBP_TUPLE* structures are placed upon return.

Figure 9-1. SO_LOOKUP_NAME *buffer*

SO_LOOKUP_ZONES, *SO_GETZONELIST*

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_LOOKUP_ZONES	Get only	1	Returns zone names from the Internet zone lists

This option requires a buffer large enough to contain a *WSH_LOOKUP_ZONES* structure at the head. Upon successful return, the space after the *WSH_LOOKUP_ZONES* structure contains the list of null-terminated zone names. The following code demonstrates how to use the *SO_LOOKUP_ZONES* option:

```
PWSH_LOOKUP_NAME     atlookup;
PWSH_LOOKUP_ZONES    zonelookup;
char                 cLookupBuffer[4096],
                     *pTupleBuffer = NULL;

atlookup = (PWSH_LOOKUP_NAME)cLookupBuffer;
zonelookup = (PWSH_LOOKUP_ZONES)cLookupBuffer;
ret = getsockopt(s, SOL_APPLETALK, SO_LOOKUP_ZONES, (char *)atlookup,
    &dwSize);
pTupleBuffer = (char *)cLookupBuffer + sizeof(WSH_LOOKUP_ZONES);
for(i = 0; i < zonelookup->NoZones; i++)
{
    printf("%3d: '%s'\n", i + 1, pTupleBuffer);
    while (*pTupleBuffer++);
}
```

SO_LOOKUP_ZONES_ON_ADAPTER, *SO_GETLOCALZONES*

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_LOOKUP_ZONES	Get only	1	Returns a list of zone names known to the given adapter name

This option is similar to *SO_LOOKUP_ZONES* except that you specify the adapter name for which you want to obtain a list of zones local to the network that that adapter is connected to. Again, you must supply a sufficiently

large buffer that has a *WSH_LOOKUP_ZONES* structure at the head. The returned list of null-terminated zone names begins in the space after the *WSH_LOOKUP_ZONES* structure. Additionally, the name of the adapter must be passed in as a UNICODE string (*WCHAR*).

SO_LOOKUP_NETDEF_ON_ADAPTER, *SO_GETNETINFO*

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_LOOKUP_NETDEF_ON_ADAPTER	Set only	1	Returns the seeded values for the network as well as the default zone

This option returns the seeded values for the network numbers and a null-terminated ANSI string containing the default zone for the network on the indicated adapter. The adapter is passed as a UNICODE (*WCHAR*) string following the structure and is overwritten by the default zone upon function return. If the network is not seeded, the network range 1-0xFFFE is returned and the null-terminated ANSI string contains the default zone "*."

SO_PAP_GET_SERVER_STATUS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_PAP_GET_SERVER_STATUS	Get only	1	Returns the PAP status from a given server

This option gets the Printer Access Protocol (PAP) status registered on the address specified in *ServerAddr* (usually obtained via an NBP lookup). The four reserved bytes correspond to the four reserved bytes in the PAP status packet. These will be in network byte order. A PAP status string can be arbitrary and is set with the option *SO_PAP_SET_SERVER_STATUS*, which we'll explain later in this chapter. The *WSH_PAP_GET_SERVER_STATUS* structure is defined as

```
#define    MAX_PAP_STATUS_SIZE        255
#define    PAP_UNUSED_STATUS_BYTES        4

typedef struct _WSH_PAP_GET_SERVER_STATUS
{
    SOCKADDR_AT            ServerAddr;
    UCHAR                  Reserved[PAP_UNUSED_STATUS_BYTES];
    UCHAR                  ServerStatus[MAX_PAP_STATUS_SIZE + 1];

} WSH_PAP_GET_SERVER_STATUS, *PWSH_PAP_GET_SERVER_STATUS;
```

The following code snippet is a quick example of how to request the PAP status. The length of the status string is the first byte of the *ServerStatus* field.

```
WSH_PAP_GET_SERVER_STATUS  status;
int                        nSize = sizeof(status);

status.ServerAddr.sat_family = AF_APPLETALK;
ret = getsockopt(s, SOL_APPLETALK, SO_PAP_GET_SERVER_STATUS,
    (char *)&status, &nSize);
```

SO_PAP_PRIME_READ

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
char []	Set only	1	This call primes a read on a PAP connection so that the sender can actually send the data.

When this option is called on a socket describing a PAP connection, it enables the remote client to send the data without the local application having called *recv* or *WSARecvEx*. After this option is set, the application can block on a *select* call and then the actual reading of the data can occur. The *optval* parameter to this call is the buffer that is to receive the data, which must be at least *MIN_PAP_READ_BUF_SIZE* (4096) bytes in length. This option allows support for nonblocking sockets on the read-driven PAP protocol. Note that for each buffer you want to read, you must make a call to *setsockopt* with the *SO_PAP_PRIME_READ* option.

SO_PAP_SET_SERVER_STATUS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
char []	Set only	1	Sets the status to be sent if another client requests the status

A client can request to obtain the PAP status by using *SO_PAP_GET_SERVER_STATUS*. This option can be used to set the status so that if clients request the PAP status, the buffer submitted to the set command will be returned on the get command. The status is a buffer of at most 255 bytes containing the status of the associated socket. If the set option is called with a null buffer, the previous status value set is erased.

SO_REGISTER_NAME

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_REGISTER_NAME	Set only	1	Registers the given name on the AppleTalk network

This option is used to register the supplied name on the AppleTalk network. If the name already exists on the network, the error *WSAEADDRINUSE* is returned. Refer to Chapter 6 for a description of the *WSH_REGISTER_NAME* structure.

SOL_IRLMP Option Level

The *SOL_IRLMP* level deals with the IrDA protocol, whose address family is *AF_IRDA*. One important thing to keep in mind when using IrDA socket options is that the implementation of infrared sockets varies among platforms. Because Windows CE first offered IR support, it does not have all the options available that were introduced later, in Windows 98 or Windows 2000. In this section, each option is followed by the platforms it is supported on.

IRLMP_9WIRE_MODE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	Gets/sets IP options within the IP header

This is another rarely used option needed to communicate with Windows 98 via IrCOMM, which is at a lower level than the level at which IrSock normally operates. In 9-wire mode, each TinyTP or IrLMP packet contains an additional 1-byte IrCOMM header. To accomplish this through the socket interface, you need to first get the maximum PDU size of an IrLMP packet with the *IRLMP_SEND_PDU_LEN* option. The socket is then put in 9-wire mode with *setsockopt* before connecting or accepting a connection. This tells the stack to add the 1-byte IrCOMM header (always set to 0) to each outgoing frame. Each *send* must be of a size less than the maximum PDU length to leave room for the added IrCOMM byte. IrCOMM is beyond the scope of this book. This option is available on Windows 98 and Windows 2000.

IRLMP_ENUMDEVICES

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
DEVICELIST	Get only	1+	Returns a list of IrDA device IDs for IR-capable devices within range

Because of the nature of infrared networking, devices capable of communicating are mobile and can move in and out of range. This option "queries" which IR devices are within range, and to connect to another device you must perform this step to obtain the device ID for each device you want to connect to.

The *DEVICELIST* structures are different on the various platforms that support IrSock because the latest platforms that added support also added functionality. Recall that Windows CE offered IrSock support first, and Windows 98 and Windows 2000 added support shortly thereafter. The *DEVICELIST* structure definition for Windows 98 and Windows 2000 is

```
typedef struct _WINDOWS_DEVICELIST
{
    ULONG                       numDevice;
    WINDOWS_IRDA_DEVICE_INFO    Device[1];
} WINDOWS_DEVICELIST, *PWINDOWS_DEVICELIST, FAR *LPWINDOWS_DEVICELIST;

typedef struct _WINDOWS_IRDA_DEVICE_INFO
{
    u_char  irdaDeviceID[4];
    char      irdaDeviceName[22];
    u_char  irdaDeviceHints1;
    u_char  irdaDeviceHints2;
    u_char  irdaCharSet;
} WINDOWS_IRDA_DEVICE_INFO, *PWINDOWS_IRDA_DEVICE_INFO,
  FAR *LPWINDOWS_IRDA_DEVICE_INFO;
```

In Windows CE, the *DEVICELIST* structure is defined as

```
typedef struct _WCE_DEVICELIST
{
    ULONG                       numDevice;
    WCE_IRDA_DEVICE_INFO        Device[1];
} WCE_DEVICELIST, *PWCE_DEVICELIST;

typedef struct _WCE_IRDA_DEVICE_INFO
{
    u_char  irdaDeviceID[4];
    char    irdaDeviceName[22];
    u_char  Reserved[2];
} WCE_IRDA_DEVICE_INFO, *PWCE_IRDA_DEVICE_INFO;
```

As you can see, the device information structure varies as well: *WCE_IRDA_DEVICE_INFO* for Windows CE and *WINDOWS_IRDA_DEVICE_INFO* for Windows 98 and Windows 2000. Each of these structures contains a field, *irdaDeviceID*, which is a 4-byte identification tag used to uniquely identify that device. You need this field to fill out the *SOCKADDR_IRDA* structure used to connect to a specific device or to manipulate or obtain an Information Access Service (IAS) entry with the options *IRLMP_IAS_SET* and *IRLMP_IAS_QUERY*.

When you call *getsockopt* to enumerate infrared devices, the *optval* parameter must be a *DEVICELIST* structure. The only requirement is that the *numDevice* field be set to 0 at first. The call to *getsockopt* does not return an error if no IR devices are discovered. After a call, the *numDevice* field should be checked to see whether it is greater than 0, which means that one or more devices were found. The *Device* field returns with a number of structures equal to the value returned in *numDevice*.

IRLMP_EXCLUSIVE_MODE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, socket connection is in exclusive mode.

This option isn't normally used by user applications, as it bypasses the TinyTP layer in the IrDA stack and communicates directly with IrLMP. If you are really interested in using this option, you should consult the IrDA specification at *http://www.irda.org*. This option is available on Windows CE and Windows 2000.

IRLMP_IAS_QUERY

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IAS_QUERY	Get only	1+	Queries IAS on a given service and class name for its attributes

This socket option is the complement of *IRLMP_IAS_SET*, as it retrieves information about a class name and its service. Before making the call to *getsockopt*, you must first fill out the *irdaDeviceID* field to the device you are querying. Set the *irdaAttribName* field to the property string on which you want to retrieve its value. The most common query would be for the LSAP-SEL number; its property string is "IrDA:IrLMP:LsapSel." Next, you need to set the *irdaClassName* field to the name of the service that the given property string applies to. Once these fields are filled, make the call to *getsockopt*. Upon success, the *irdaAttribType* field indicates which field in the union to obtain the information from. Use the identifiers in Table 9-2 to decode this entry. The most common error is *WSASERVICE_NOT_FOUND*, which is returned when the given service is not found on that device. This option is available on Windows CE, Windows 98, and Windows 2000.

IRLMP_IAS_SET

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IAS_QUERY	Set only	1+	Sets an attribute value for a given class name and attribute

IAS is a dynamic service registration entity that can be queried and modified. The *IRLMP_IAS_SET* option allows you to set a single attribute for a single class within the local IAS. As with *IRLMP_ENUMDEVICES*, there are separate structures for Windows CE and for Windows 98 and Windows 2000. The structures for Windows 98 and Windows 2000 are

```
typedef struct _WINDOWS_IAS_QUERY
{
    u_char     irdaDeviceID[4];
    char       irdaClassName[IAS_MAX_CLASSNAME];
    char       irdaAttribName[IAS_MAX_ATTRIBNAME];
    u_long     irdaAttribType;
    union
    {
        LONG   irdaAttribInt;
        struct
        {
            u_long   Len;
            u_char   OctetSeq[IAS_MAX_OCTET_STRING];
        } irdaAttribOctetSeq;
        struct
        {
            u_long   Len;
            u_long   CharSet;
            u_char   UsrStr[IAS_MAX_USER_STRING];
        } irdaAttribUsrStr;
    } irdaAttribute;
} WINDOWS_IAS_QUERY, *PWINDOWS_IAS_QUERY, FAR *LPWINDOWS_IAS_QUERY;
```

The IAS query structure for Windows CE is

```
typedef struct _WCE_IAS_QUERY
{
    u_char     irdaDeviceID[4];
    char       irdaClassName[61];
    char       irdaAttribName[61];
    u_short    irdaAttribType;
    union
    {
        int    irdaAttribInt;
        struct
        {
            int      Len;
            u_char   OctetSeq[1];
            u_char   Reserved[3];
        } irdaAttribOctetSeq;
        struct
        {
            int      Len;
            u_char   CharSet;
            u_char   UsrStr[1];
            u_char   Reserved[2];
        } irdaAttribUsrStr;
    } irdaAttribute;
} WCE_IAS_QUERY, *PWCE_IAS_QUERY;
```

Table 9-2 provides the different constants for the *irdaAttribType* field, which indicates which type the attribute belongs to. The last two entries are not values that you can set, but values that a call to *getsockopt* with an *IRLMP_IAS_QUERY* socket option can return in the *irdaAttribType* field. These are included in the table for the sake of completeness.

Table 9-2. *IAS attribute types*

irdaAttribType *Value*	*Field to Set*
IAS_ATTRIB_INT	*IrdaAttribInt*
IAS_ATTRIB_OCTETSEQ	*IrdaAttribOctetSeq*
IAS_ATTRIB_STR	*IrdaAttribUsrStr*
IAS_ATTRIB_NO_CLASS	None
IAS_ATTRIB_NO_ATTRIB	None

In order to set a value, you must fill in *irdaDeviceID* to the IR device on which to modify the IAS entry. Also, *irdaAttribName* must be set to the class on which to set the attribute, while *irdaClassName* usually refers to the service on which to set the attribute. Remember that with IrSock, socket servers are services registered with IAS that have an associated LSAP-SEL number used by clients to connect to the server. The LSAP-SEL number is an attribute associated with that service. To modify the LSAPSEL number in the service's IAS entry, set the *irdaDeviceID* field to the device ID on which the service is running. Set the *irdaAttribName* field to the property string "IrDA:IrLMP:LsapSel" and the *irdaClassName* field to the name of the service (for example, "MySocketServer"). From there, set *irdaAttribType* to *IAS_ATTRIB_INT* and *irdaAttribInt* to the new LSAP-SEL number. Of course, changing the service's LSAP-SEL number is a bad idea, but this example is for illustration only.

IRLMP_IRLPT_MODE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, socket is configured to communicate to IR capable printers.

It is possible to connect to an infrared printer using Winsock and send data to be printed. This is accomplished by putting the socket in IRLPT mode before establishing the connection. Simply pass the Boolean value *TRUE* to this option after socket creation. You can use the option *IRLMP_ENUMDEVICES* to find infrared-capable printers within range. Note that some legacy IR printers do not register themselves with IAS; you might need to connect to them directly using the "LSAP-SEL-*xxx*" identifier. See Chapter 6 and its discussion of IrSock for more details on bypassing IAS. This option is available on Windows CE and Windows 2000.

IRLMP_SEND_PDU_LEN

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Get only	1+	Gets the maximum PDU length

This option retrieves the maximum Protocol Data Unit (PDU) size needed when using the option *IRLMP_9WIRE_MODE*. See the description of *IRLMP_9WIRE_MODE* for more information about this option, which is available on Windows CE and Windows 2000.

IPPROTO_IP Option Level

The socket options on the *IPPROTO_IP* level pertain to attributes specific to the IP protocol, such as modifying certain fields in the IP header and adding a socket to an IP multicast group. Many of these options are declared in both Winsock.h and Winsock2.h with different values. Note that if you load Winsock 1, you must include the correct header and link with Wsock32.lib. Likewise for Winsock 2, you should include the Winsock 2 header file and link with Ws2_32.lib. This is especially relevant to multicasting, which is available under both versions. Multicasting is supported on all Win32 platforms except Windows CE, in which it is available on versions 2.1 and later.

IP_OPTIONS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
char []	Both	1+	Gets/sets IP options within the IP header

This flag allows you to set various IP options within the IP header. Some of the possible options are

- Security and handling restrictions RFC 1108.
- Record route Each router adds its IP address to the header (see the ping sample in Chapter 13).
- Timestamp Each router adds its IP address and time.
- Loose source routing The packet is required to visit each IP address listed in the option header.
- Strict source routing The packet is required to visit *only* those IP addresses listed in the option header.

Be aware that hosts and routers do not support all of these options.

When setting an IP option, the data that you pass into the *setsockopt* call follows the structure shown in Figure 9-2. The IP option header can be up to 40 bytes in length.

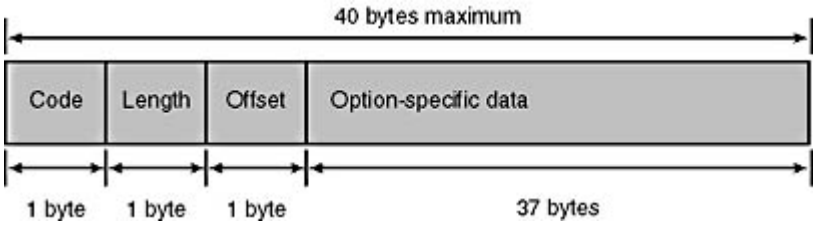

Figure 9-2. *IP option header format*

The code field indicates what type of IP option is present. For example, the value 0x7 represents the record route option. Length is simply the length of the option header, while offset is the offset value into the header where the data portion of the header begins. The data portion of the header is specific to the particular option. In the following code snippet, we set up the record route option. Notice that we declare a structure (*struct ip_option_hdr*) that contains the first three option values (code, length, offset), and then we declare the option-specific data as an array of nine unsigned long integers, as the data to be recorded is up to nine IP addresses. Remember that the maximum size of the IP option header is 40 bytes; however, our structure occupies only 39 bytes. The system will pad the header to a multiple of a 32-bit word for you (up to 40 bytes).

```
struct ip_option_hdr
{
    unsigned char    code;
    unsigned char    length;
    unsigned char    offset;
    unsigned long    addrs[9];
} opthdr;

...

ZeroMemory((char *)&opthdr, sizeof(opthdr));
opthdr.code = 0x7;
opthdr.length = 39;
opthdr.offset = 4;  // Offset to first address (addrs)
ret = setsockopt(s, IPPROTO_IP, IP_OPTIONS, (char *)&opthdr,
        sizeof(opthdr));
```

Once the option is set, it applies to any packets sent on the given socket. At any pointer thereafter, you can call *getsockopt* with *IP_OPTIONS* to retrieve which options were set; however, this will not return any data filled into the option-specific buffers. In order to retrieve the data set in the IP options, either the socket must be created as a raw socket (*SOCK_RAW*) or the *IP_HDRINCL* option should be set—in which case, the IP header is returned along with data after a call to a Winsock receive function.

IP_HDRINCL

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	2+	If *TRUE*, IP header is submitted with data to be sent and returned from data that is read.

Setting the *IP_HDRINCL* option to *TRUE* causes the send function to include the IP header ahead of the data it's sending and causes the receive function to include the IP header as part of the data. Thus, when you call a Winsock send function, you must include the entire IP header ahead of the data and fill each field of the IP header correctly. Figure 9-3 shows what the IP header should look like. This option is available only on Windows 2000.

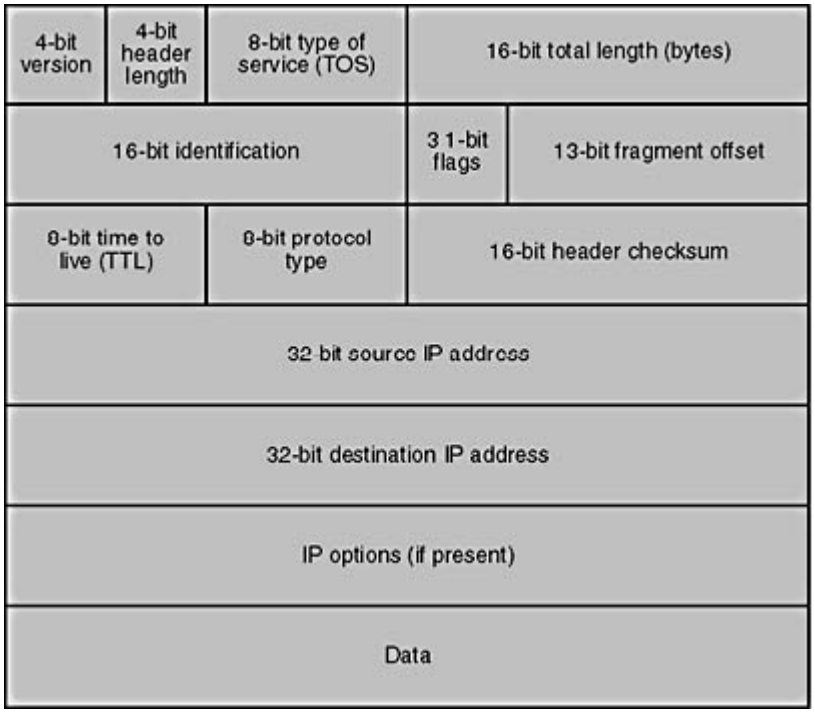

Figure 9-3. *The IP header*

The first field of the header is the IP version, which is currently version 4. The header length is the number of 32-bit words in the header. An IP header must always be a multiple of 32 bits. The next field is the type of service field. Consult the *IP_TOS* socket option, discussed next, for additional information. The total length field is the length, in bytes, of the IP header and data. The identification field is a unique value used to identify each IP packet sent. Normally, the system increments this value with each packet sent. The flags and fragmentation offset fields are used when IP packets are fragmented into smaller packets. The time to live field, or TTL, limits the number of routers through which the packet can pass. Each time a router forwards the packet, the TTL is decremented by 1. Once the TTL is 0, the packet is dropped. This limits the amount of time a packet can be live on the network. The protocol field is used to demultiplex incoming packets. Some of the valid protocols that use IP addressing are TCP, UDP, IGMP, and ICMP. The checksum is the 16-bit one's complement sum of the header. It is calculated over the header only and not the data. The next two fields are the 32-bit IP source and destination addresses. The IP options field is a variable length field that contains optional information, usually regarding security or routing.

The easiest way to include an IP header with the data that you are sending is to define a structure that contains the IP header and the data, and pass the structure into the Winsock *send* call. See Chapter 13 for more details and an example of this option. This option works only on Windows 2000.

IP_TOS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	IP type of service

The type of service (TOS) is a field present in the IP header that is used to signify certain characteristics about a packet. The field is 8 bits long and is broken into three parts: a 3-bit precedence field (which is ignored), a 4-bit TOS field, and the remaining bit (which must be 0). The 4 TOS bits are minimize delay, maximize throughput, maximize reliability, and minimize monetary costs. Only 1 bit can be set at a time. All 4 bits being 0 implies normal service. RFC 1340 specifies the recommended bits to set for various standard applications such as TCP, SMTP, NNTP, and so on. Additionally, RFC 1349 contains some corrections to the original RFC.

Interactive applications—such as Rlogin or Telnet—might want to minimize delay. Any kind of file transfer—such as FTP—is interested in maximum throughput. Maximum reliability is used by network management (SNMP) and routing protocols. Finally, Usenet news (NNTP) is an example of minimizing monetary costs. The *IP_TOS* option is not available on Windows CE.

There is an additional issue when you attempt to set the TOS bits on a QOS-enabled socket. Because IP precedence is used by QOS to differentiate levels of service, it is undesirable to allow developers the ability to change these values. As a result, when you call *setsockopt* with *IP_TOS* on a QOS-enabled socket, the QOS service provider intercepts the call to verify whether the change can take place. See Chapter 12 for more information about QOS.

IP_TTL

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	IP time-to-live parameter

The time-to-live (TTL) field is present in an IP header. Datagrams use the TTL field to limit the number of routers through which the datagram can pass. The purpose of this limitation is to prevent routing loops in which datagrams can spin in circles forever. The idea behind this is that each router that the datagram passes through decrements the datagram's TTL value by 1. When the value equals 0, the datagram is discarded. This option is not available on Windows CE.

IP_MULTICAST_IF

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
unsigned long	Both	1+	Gets/sets the local interface for multicast data to be sent from

The IP multicast interface (IF) option sets the local interface from which any multicast data sent by the local machine will be sent. This option is only of interest on machines that have more than one connected network interface (network card, modem, and so on). The *optval* parameter should be an unsigned long integer representing the binary IP address of the local interface. The function *inet_addr* can be used to convert a string IP dotted decimal address to an unsigned long integer, as in the following sample:

```
DWORD    mcastIF;

// First join socket s to a multicast group
mcastIF = inet_addr("129.113.43.120");
ret = setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, (char *)&mcastIF,
    sizeof(mcastIF));
```

IP_MULTICAST_TTL

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	Gets/sets the time to live on multicast packets for this socket

Similar to the IP TTL, this option performs the same function except that it applies only to multicast data sent using the given socket. Again, the purpose of the TTL is to prevent routing loops, but in the case of multicasting, setting the TTL narrows the scope of how far the data will travel. Therefore, multicast group members must be within "range" to receive datagrams. The default TTL value for multicast datagrams is 1.

IP_MULTICAST_LOOP

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, data sent to a multicast address will be echoed to the socket's incoming buffer.

By default, when you send IP multicast data, the data will be looped back to the sending socket if it is also a member of that multicast group. If you set this option to *FALSE*, any data sent will not be posted to the incoming data queue for the socket.

IP_ADD_MEMBERSHIP

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
struct ip_mreq	Set only	1+	Adds the socket to the given IP group membership

This option is the Winsock 1 method of adding a socket to an IP multicast group. This is done by creating a socket of address family *AF_INET* and the socket type *SOCK_DGRAM* with the *socket* function. To add the socket to a multicast group, use the following structure:

```
struct ip_mreq
{
    struct in_addr  imr_multiaddr;
    struct in_addr  imr_interface;
};
```

In the *ip_mreq* structure, *imr_multiaddr* is the binary address of the multicast group to join, while *imr_interface* is the local interface that multicast data should be sent out and received on. See Chapter 11 for more information about valid multicast addresses. The *imr_interface* field is either the binary IP address of a local interface or the value *INADDR_ANY*, which can be used to select the default interface.

IP_DROP_MEMBERSHIP

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
struct ip_mreq	Set only	1+	Removes the socket from the given IP group membership

This option is the opposite of *IP_ADD_MEMBERSHIP*. By calling this option with an *ip_mreq* structure that contains the same values used when joining the given multicast group, the socket *s* will be removed from the given group. Again, Chapter 11 contains much more detailed information on IP multicasting.

IP_DONTFRAGMENT

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, do not fragment IP datagrams.

This flag tells the network not to fragment the IP datagram during transmission. However, if the size of the IP datagram exceeds the maximum transmission unit (MTU) and the IP don't fragment flag is set within the IP header, the datagram will be dropped and an ICMP error message ("fragmentation needed but don't fragment bit set") will be returned to the sender. This option is not available on Windows CE.

IPPROTO_TCP Option Level

There is only one option belonging to the *IPPROTO_TCP* level. The option is valid only for sockets that are stream sockets (*SOCK_STREAM*) and belong to family *AF_INET*. This option is available on all versions of Winsock and is supported on all Win32 platforms.

TCP_NODELAY

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, the Nagle algorithm is disabled on the socket.

In order to increase performance and throughput by minimizing overhead, the system implements the Nagle algorithm. When an application requests to send a chunk of data, the system might hold on to that data for a while and wait for other data to accumulate before actually sending it on the wire. Of course, if no other data accumulates in a given period of time, the data will be sent regardless. This results in more data in a single TCP packet, as opposed to smaller chunks of data in multiple TCP packets. The overhead is that the TCP header for each packet is 20 bytes long. Sending a couple bytes here and there with a 20-byte header is wasteful. The other part of this algorithm is the delayed acknowledgments. Once a system receives TCP data it must send an ACK to the peer. However, the host will wait to see whether it has data it is sending to the peer so that it can piggyback the ACK on the data to be sent—resulting in one less packet on the network.

The purpose of this option is to disable the Nagle algorithm, as its behavior can be detrimental in a few cases. This algorithm can adversely affect any network application that sends relatively small amounts of data and expects a timely response. A classic example is Telnet. Telnet is an interactive application that allows the user to log on to a remote machine and send it commands. Typically the user hits only a few keystrokes per second. The Nagle algorithm would make such a session seem sluggish and unresponsive.

NSPROTO_IPX Option Level

These socket options are Microsoft-specific extensions to the Window IPX/SPX Windows Sockets interface, provided for use as necessary for compatibility with existing applications. They are otherwise not recommended for use, as they are guaranteed to work only over the Microsoft IPX/SPX stack. An application that uses these extensions might not work over other IPX/SPX implementations. These options are defined in WSNwLink.h, which should be included after Winsock.h and Wsipx.h.

IPX_PTYPE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	Gets/sets the IPX packet type

This option sets or gets the IPX packet type. The value specified in the *optval* argument will be set as the packet type on every IPX packet sent from this socket. The *optval* parameter is an integer.

IPX_FILTERPTYPE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	Gets/sets the IPX packet type to filter on

This option gets or sets the receive filter packet type. Only IPX packets with a packet type equal to the value specified in the *optval* argument are returned on any receive call; packets with a packet type that does not match are discarded.

IPX_STOPFILTERPTYPE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Set only	1+	Removes the filter on the given IPX packet

You can use this option to stop filtering on packet types that are set with the *IPX_FILTERPTYPE* option.

IPX_DSTYPE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	Sets/gets the value of the datastream field in the SPX header

This option gets or sets the value of the datastream field in the SPX header of every packet sent.

IPX_EXTENDED_ADDRESS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, enables extended addressing on IPX packets

This option enables or disables extended addressing. On sends, it adds the element *unsigned char sa_ptype* to the *SOCKADDR_IPX* structure, making the total length of the structure 15 bytes. On receives, the option adds both the *sa_ptype* and *unsigned char sa_flags* elements to the *SOCKADDR_IPX* structure, making the total length 16 bytes. The current bits defined in *sa_flags* are

- 0x01 The received frame was sent as a broadcast.
- 0x02 The received frame was sent from this machine.

IPX_RECVHDR

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, returns IPX header with receive call

If this option is set to true, any Winsock receive call returns the IPX header along with the data.

IPX_MAXSIZE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Get only	1+	Returns the maximum IPX datagram size

Calling *getsockopt* with this option returns the maximum IPX datagram size possible.

IPX_ADDRESS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IPX_ADDRESS_DATA	Get only	1+	Returns information regarding an IPX-capable adapter

This option queries for information about a specific adapter that IPX is bound to. In a system with *n* adapters, the adapters are numbered 0 through *n* - 1. To find the number of IPX-capable adapters on the system, use the *IPX_MAX_ADAPTER_NUM* option with *getsockopt*, or call *IPX_ADDRESS* with increasing values of *adapternum* until it fails. The *optval* parameter points to an *IPX_ADDRESS_DATA* structure defined as

```
typedef struct _IPX_ADDRESS_DATA
{
    INT     adapternum;  // Input: 0-based adapter number
    UCHAR   netnum[4];   // Output: IPX network number
    UCHAR   nodenum[6];  // Output: IPX node address
    BOOLEAN wan;         // Output: TRUE = adapter is on a WAN link
    BOOLEAN status;      // Output: TRUE = WAN link is up (or adapter
                         // is not WAN)
    INT     maxpkt;      // Output: max packet size, not including IPX
                         // header
    ULONG   linkspeed;   // Output: link speed in 100 bytes/sec
                         // (i.e., 96 == 9600 bps)
} IPX_ADDRESS_DATA, *PIPX_ADDRESS_DATA;
```

IPX_GETNE TINFO

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IPX_NETNUM_DATA	Get only	1+	Returns information regarding a specific IPX network number

This option obtains information about a specific IPX network number. If the network is in IPX's cache, the option returns the information directly; otherwise, it issues RIP requests to find it. The *optval* parameter points to a valid *IPX_NETNUM_DATA* structure defined as

```
typedef struct _IPX_NETNUM_DATA
{
    UCHAR  netnum[4];  // Input: IPX network number
    USHORT hopcount;   // Output: hop count to this network, in machine
                       // order
    USHORT netdelay;   // Output: tick count to this network, in machine
                       // order
    INT    cardnum;    // Output: 0-based adapter number used to route
                       // to this net; can be used as adapternum input
                       // to IPX_ADDRESS
    UCHAR  router[6];  // Output: MAC address of the next hop router,
                       // zeroed if the network is directly attached
} IPX_NETNUM_DATA, *PIPX_NETNUM_DATA;
```

IPX_GETNETINFO_NORIP

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IPX_NETNUM_DATA	Both	1+	If *TRUE*, do not fragment IP datagrams.

This option is similar to *IPX_GETNETINFO* except that it does not issue RIP requests. If the network is in IPX's cache, it returns the information; otherwise, it fails. (See also *IPX_RERIPNETNUMBER*, which always issues RIP requests.) Like *IPX_GETNETINFO*, this option requires passing an *IPX_NETNUM_DATA* structure as the *optval* parameter.

IPX_SPXGETCONNECTIONSTATUS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IPX_SPXCONNSTATUS_DATA	Get only	1+	Returns information regarding a connected SPX socket

This option returns information on a connected SPX socket. The *optval* parameter points to an *IPX_SPXCONNSTATUS_DATA* structure defined below. All numbers are in network (high-low) byte order.

```
typedef struct _IPX_SPXCONNSTATUS_DATA
{
    UCHAR  ConnectionState;
    UCHAR  WatchDogActive;
    USHORT LocalConnectionId;
    USHORT RemoteConnectionId;
    USHORT LocalSequenceNumber;
    USHORT LocalAckNumber;
    USHORT LocalAllocNumber;
    USHORT RemoteAckNumber;
    USHORT RemoteAllocNumber;
    USHORT LocalSocket;
    UCHAR  ImmediateAddress[6];
    UCHAR  RemoteNetwork[4];
    UCHAR  RemoteNode[6];
    USHORT RemoteSocket;
    USHORT RetransmissionCount;
    USHORT EstimatedRoundTripDelay; /* In milliseconds */
    USHORT RetransmittedPackets;
    USHORT SuppressedPacket;
} IPX_SPXCONNSTATUS_DATA, *PIPX_SPXCONNSTATUS_DATA;
```

IPX_ADDRESS_NOTIFY

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IPX_ADDRESS_DATA	Get only	1+	Asynchronously notifies when the status of an IPX adapter changes

This option submits a request to be notified when the status of an adapter that IPX is bound to changes, which typically occurs when a WAN line goes up or down. This option requires the caller to submit an *IPX_ADDRESS_DATA* structure as the *optval* parameter. The exception, however, is that the *IPX_ADDRESS_DATA* structure is followed immediately by a handle to an unsignaled event. The following pseudo-code illustrates one method for calling this option.

```
char  buff[sizeof(IPX_ADDRESS_DATA) + sizeof(HANDLE)];
IPX_ADDRESS_DATA *ipxdata;
HANDLE *hEvent;

ipxdata = (IPX_ADDRESS_DATA *)buff;
hEvent = (HANDLE *)(buff + sizeof(IPX_ADDRESS_DATA));
ipxdata->adapternum = 0;  // Set to the appropriate adapter
*hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
setsockopt(s, NSPROTO_IPX, IPX_ADDRESS_NOTIFY, (char *)buff,
sizeof(buff));
```

When the *getsockopt* query is submitted, it completes successfully. However, the *IPX_ADDRESS_DATA* structure pointed to by *optval* will not be updated at that point. Instead, the request is queued internally inside the transport, and when the status of an adapter changes, IPX locates a queued *getsockopt* query and fills in all the fields in the *IPX_ADDRESS_DATA* structure. It then signals the event pointed to by the handle in the *optval* buffer. If multiple *getsockopt* calls are submitted at once, different events must be used. The event is used because the call needs to be asynchronous; *getsockopt* does not currently support this.

> WARNING
>
> In the current implementation, the transport signals only one queued query for each status change. Therefore, only one service that uses a queued query should run at once.

IPX_MAX_ADAPTER_NUM

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Get only	1+	Returns the number of IPX adapters present

This option returns the number of IPX-capable adapters present on the system. If this call returns *n* adapters, the adapters are numbered 0 through *n* - 1.

IPX_RERIPNETNUMBER

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IPX_NETNUM_DATA	Get only	1+	Returns information regarding a network number

This option is related to IPX_GETNETINFO except that it forces IPX to reissue RIP requests even if the network is in its cache (but not if it is directly attached to that network). Like *IPX_GETNETINFO*, it requires passing an *IPX_NETNUM_DATA* structure as the *optval* parameter.

IPX_RECEIVE_BROADCAST

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Set only	1+	If *TRUE*, do not receive broadcast IPX packets.

By default, an IPX socket is capable of receiving broadcast packets. Applications that do not need to receive broadcast packets should set this option to *FALSE*, which can cause better system performance. Note, however, that setting the option to *FALSE* does not necessarily cause broadcasts to be filtered for the application.

IPX_IMMEDIATESPXACK

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, do not delay sending ACKs on SPX connections.

If you set this option to *true*, acknowledgement packets will not be delayed for SPX connections. Applications that do not tend to have back-and-forth traffic over SPX should set this—it increases the number of ACKs sent but prevents the appearance of slow performance as a result of delayed acknowledgments.

IOCTLSOCKET and *WSAIOCTL*

The socket *ioctl* functions are used to control the behavior of I/O upon the socket, as well as obtain information about I/O pending on that socket. The first function, *ioctlsocket* , originated in the Winsock 1 specification and is declared as

```
int ioctlsocket (
    SOCKET s,
    long cmd,
    u_long FAR *argp
);
```

The parameter *s* is the socket descriptor to act upon, while *cmd* is a predefined flag for the I/O control command to execute. The last parameter, *argp* , is a pointer to a variable specific to the given command. When each command is described, the type of the required variable is given. Winsock 2 introduced a new *ioctl* function that adds quite a few new options. First, it breaks the single *argp* parameter into a set of input parameters for values passed into the function and a set of output parameters used to return data from the call. Additionally, the function call can use overlapped I/O. This function is *WSAIoctl* , which is defined as

```
int WSAIoctl(
    SOCKET s,
    DWORD dwIoControlCode,
    LPVOID lpvInBuffer,
    DWORD cbInBuffer,
    LPVOID lpvOutBuffer,
    DWORD cbOutBuffer,
    LPDWORD lpcbBytesReturned,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

The first two parameters are the same as those in *ioctlsocket* . The second two parameters, *lpvInBuffer* and *cbInBuffer* , describe the input parameters. The *lpvInBuffer* parameter is a pointer to the value passed in, while *cbInBuffer* is the size of that data in bytes. Likewise, *lpvOutBuffer* and *cbOutBuffer* are used for any data returned from the call. The *lpvOutBuffer* parameter points to a data buffer in which any information returned is placed. The *cbOutBuffer* parameter is the size in bytes of the buffer passed in as *lpvOutBuffer* . Note that some calls might use only input or output parameters, while others will use both. The seventh parameter, *lpcbBytesReturned* , is the number of bytes actually returned. The last two parameters, *lpOverlapped* and *lpCompletionRoutine* , are used when calling this function with overlapped I/O. Consult Chapter 8 for detailed information on using overlapped I/O.

Standard Ioctl Commands

These three ioctl commands are the most common and are carryovers from the Unix world. They are available on all Win32 platforms. Also, these three commands can be called using either *ioctlsocket* or *WSAIoctl* .

FIONBIO

ioctlsocket/WSAIoctl
unsigned
None
1+
Puts socket in nonblocking mode

Which Function?	Input	Output	Winsock Version	Description

This command enables or disables nonblocking mode on socket *s* . By default, all sockets are blocking sockets upon creation. When you call *ioctlsocket* with the *FIONBIO* ioctl command, set *argp* to pass a pointer to an unsigned long integer whose value is nonzero if nonblocking mode is to be enabled. The value 0 places the socket in blocking mode. If you use *WSAIoctl* instead, simply pass the unsigned long integer in as the *lpvInBuffer* parameter.

Calling the *WSAAsyncSelect* or *WSAEventSelect* function automatically sets a socket to nonblocking mode. If either of these functions has been called, any attempt to set the socket back to blocking mode fails with *WSAEINVAL* . To set the socket back to blocking mode, an application must first disable *WSAAsyncSelect* by calling *WSAAsyncSelect* with the *lEvent* parameter equal to 0, or disable *WSAEventSelect* by calling *WSAEventSelect* with the *lNetworkEvents* parameter equal to 0.

FIONREAD

Both
None
unsigned long
1+
Returns the amount of data to be read on the socket

Which Function?	Input	Output	Winsock Version	Description

This command determines the amount of data that can be read atomically from the socket. For *ioctlsocket* , the *argp* value returns with an unsigned integer that will contain the number of bytes to be read. When using *WSAIoctl* , the unsigned integer is returned in *lpvOutBuffer* . If socket *s* is stream-oriented (*SOCK_STREAM*), *FIONREAD* returns the total amount of data that can be read in a single receive call. Remember that using this or any other message-peeking mechanism is not always guaranteed to return the correct amount. When this ioctl command is used on a datagram socket (*SOCK_DGRAM*), the return value is the size of the first message queued on the socket.

SIOCATMARK

Both
None
BOOL
1+
Determines whether out-of-band data has been read

Which Function?	Input	Output	Winsock Version	Description

When a socket has been configured to receive out-of-band (OOB) data and has been set to receive this OOB data in line (by setting the *SO_OOBINLINE* socket option), this ioctl command returns a Boolean value indicating *TRUE* if the OOB data is to be read next. Otherwise, *FALSE* is returned and the next receive operation returns all or some of the data that precedes the OOB data. For *ioctlsocket* , *argp* returns with a pointer to a Boolean variable, while for *WSAIoctl* , the pointer to the Boolean variable returns in *lpvOutBuffer* . Remember that a receive call will never mix OOB data and normal data in the same call. Refer back to Chapter 7 for more information on OOB data.

Other Ioctl Commands

These ioctl commands are specific to Winsock 2 except for those dealing with SSL, which are available only on Windows CE. If you examine the Winsock 2 headers, you might actually see other ioctl commands declared; however, the ioctls listed in this section are the only ones that are meaningful or available to a user's application. Additionally, as you will see, not all ioctl commands work on all (or any) Win32 platforms, but of course this could change with operating system updates. For Winsock 2, a majority of these commands are defined in Winsock2.h. Some of the newer, Windows 2000-specific, ioctls are defined in Mstcpip.h.

SIO_ENABLE_CIRCULAR_QUEUEING

WSAIoctl
BOOL
BOOL

2+
If the incoming buffer queue overflows, discard oldest message first.

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command controls how the underlying service provider handles incoming datagram messages when the queues are full. By default, when the incoming queue is full, any datagram messages subsequently received are dropped. When this option is set to *TRUE* , it indicates that the newly arrived messages should never be dropped as a result of buffer overflow; instead, the oldest message in the queue should be discarded in order to make room for the newly arrived message. This command is valid only for sockets associated with unreliable, message-oriented, protocols. If this ioctl command is used on a socket of another type (such as a stream-oriented protocol socket), or if the service provider doesn't support the command, the error *WSAENOPROTOOPT* is returned. This option is supported only on Windows NT and Windows 2000.

This ioctl command can be used either to set circular queuing on or off or to query the current state of the option. When you are setting the option, only the input parameters need to be used. When you are querying the current value of the option, only the output *BOOL* parameter needs to be supplied.

SIO_FIND_ROUTE

WSAIoctl
SOCKADDR
BOOL
2+
Verifies that a route to the given address exists

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command is used to check whether a particular address can be reached via the network. The *lpvInBuffer* parameter points to a *SOCKADDR* structure for the given protocol. If the address already exists in the local cache, it is invalidated. For IPX, this call initiates an IPX *GetLocalTarget* call that queries the network for the given remote address. Unfortunately, the Microsoft provider for current Win32 platforms does not implement this ioctl command.

SIO_FLUSH

WSAIoctl
None
None
2+
Determines whether OOB data has been read

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command discards the current contents of the sending queue associated with the given socket. There are no input or output parameters for this option. Currently only Windows 2000 and Windows NT 4 Service Pack 4 implement this option.

SIO_GET_BROADCAST_ADDRESS

WSAIoctl
None
SOCKADDR
2+
Returns a broadcast address for the address family of the socket

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command returns a *SOCKADDR* structure (via *lpvOutBuffer*) that contains the broadcast address for the address family of socket *s* that can be used in *sendto* or *WSASendTo* . This ioctl works only on Windows NT and Windows 2000. Windows 95 and Windows 98 return *WSAEINVAL* .

SIO_GET_EXTENSION_FUNCTION_POINTER

WSAIoctl
GUID
function pointer?
2+
Retrieves a function pointer specific to underlying provider

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command is used to obtain functions that are provider-specific but are not part of the Winsock specification. If a provider chooses, it can make functions available to programmers through this ioctl command by assigning each function a GUID. Then an application can obtain a pointer to this function by using the *SIO_GET_EXTENSION_FUNCTION_POINTER* ioctl. The header file Mswsock.h defines those Winsock functions that Microsoft has added, including their GUIDs. For example, to query whether the installed Winsock provider supports the *TransmitFile* function, you can query the provider by using its GUID, which is given by the following define:

```
#define WSAID_TRANSMITFILE \
   {0xb5367df0,0xcbac,0x11cf,{0x95,0xca,0x00,0x80,0x5f,0x48,0xa1,0x92}}
```

Once you obtain the function pointer for an extension function such as *TransmitFile* , you can call it directly without having to link your application to the Mswsock.lib library. This will actually reduce one intermediate function call that is made in Mswsock.lib.

You can look through Mswsock.h for other Microsoft-specific extensions that have these GUIDs defined for them. This ioctl command is an important part of developing a layered service provider. See Chapter 14 for more details of the service provider interface.

SIO_CHK_QOS

WSAIoctl
DWORD
DWORD
2+
Sets the QOS attributes for the given socket

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command can be used to check the status of six states within QOS and is currently supported only on Windows 2000. Six different flags correspond to these states: *ALLOWED_TO_SEND_DATA* , *ABLE_TO_RECV_RSVP* , *LINE_RATE* , *LOCAL_TRAFFIC_CONTROL* , *LOCAL_QOSABILITY* , and *END_TO_END_QOSABILITY* .

The first flag, *ALLOWED_TO_SEND_DATA* , is used once QOS levels are set on a socket using *SIO_SET_QOS* but before any RSVP reservation request (RESV) message has been received. Receiving a RESV message indicates that the desired bandwidth requirements have been allocated to your flow. Prior to receiving the RESV message, the flow corresponding to the socket is given only best-effort service. The RSVP protocol and reservation of network resources are covered in greater detail in Chapter 12 . Use the *ALLOWED_TO_SEND_DATA* flag to see whether the current best-effort service is sufficient for the levels of QOS requested by *SIO_SET_QOS* . The return value will be either 1—meaning that the current best-effort bandwidth is sufficient—or 0, meaning that the bandwidth cannot accommodate the requested levels. If the *ALLOWED_TO_SEND_DATA* flag returns 0, the sending application should wait until a RESV message is received before sending data.

The second flag, *ABLE_TO_RECV_RSVP* , indicates whether the host is able to receive and process RSVP messages on the interface that the given socket is bound to. The return value is either 1 or 0, corresponding to whether RSVP messages can or cannot be received, respectively.

The next flag, *LINE_RATE* , returns the best-effort line rate in kilobits per second (kbps). If the line rate is not known, the value *INFO_NOT_AVAILABLE* is returned.

The last three flags indicate whether certain capabilities exist on the local machine or the network. All three options return 1 to indicate the option is supported, 0 if it is not supported, or *INFO_NOT_AVAILABLE* if there is no way to check. *LOCAL_TRAFFIC_CONTROL* is used to determine whether the Traffic Control component is installed and available on the machine. *LOCAL_QOSABILITY* determines whether QOS is supported on the local machine. Finally *END_TO_END_QOSABILITY* indicates whether the local network is QOS-enabled.

SIO_GET_QOS

WSAIoctl
None
QOS
2+
Returns the *QOS* structure associated with the socket

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command retrieves the *QOS* structure associated with the socket. The supplied buffer must be large enough to contain the whole structure, which in some cases is larger than *sizeof(QOS)* , as the structure might contain provider-specific information. For more information on QOS, see Chapter 12 . If this ioctl command is used on a socket whose address family does not support QOS, the error *WSAENOPROTOOPT* is returned. This option and *SIO_SET_QOS* are available only on platforms that provide a QOS-capable transport, such as Windows 98 and Windows 2000.

SIO_SET_QOS

WSAIoctl
QOS
None
2+
Sets the QOS attributes for the given socket

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command is the companion to *SIO_GET_QOS* . The input parameter for this call is a *QOS* structure that defines the bandwidth requirements for this socket. This call does not return any output values. See Chapter 12 for more information about QOS. This option and *SIO_GET_QOS* are available only on those platforms that provide a QOS-capable transport, such as Windows 98 and Windows 2000.

SIO_MULTIPOINT_LOOPBACK

WSAIoctl
BOOL
BOOL
2+
Sets/gets whether multicast data will be looped back to the socket

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

When sending multicast data, the default behavior is to have any data sent to the multicast group posted as incoming data on the socket's receive queue. Of course, this loopback is in effect only if the socket is also a member of the multicast group that it is sending to. Currently this loopback behavior is seen only in IP multicasting and is not present in ATM multicasting. To disable this loopback, pass a Boolean variable with the value *FALSE* into the input parameter *lpvInBuffer* . To obtain the current value of this option, leave the input value as *NULL* and supply a Boolean variable as the output parameter.

SIO_MULTICAST_SCOPE

WSAIoctl
int
int
2+
Gets/sets the time-to-live (TTL) value for multicast data

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command controls the lifetime, or scope, of multicast data. The scope is the number of routed network segments that data is allowed to traverse; by default, the value is only 1. When a multicast packet hits a router, the TTL value is decremented by 1. Once the TTL reaches 0, the packet is discarded. To set the value, pass an integer with the desired TTL as *lpvInBuffer* ; otherwise, to simply obtain the current TTL value, call *WSAIoctl* with the *lpvOutBuffer* pointing to an integer.

SIO_KEEPALIVE_VALS

WSAIoctl
tcp_keepalive
tcp_keepalive
2+
Sets the TCP keepalive active on a per-connection basis

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command allows setting the TCP keepalive active on a per-connection basis and allows you to specify the keepalive interval. The socket option *SO_KEEPALIVE* also enables TCP keepalives, but the interval on which they are sent is set in the Registry. Changing the Registry value changes the keepalive interval for all processes on the machine. This ioctl command allows you to set the interval on a per-socket basis. To set the keepalive active on the given connected socket, initialize a *tcp_keepalive* structure and pass it as the input buffer. The structure is defined as follows:

```
struct tcp_keepalive
{
    u_long    onoff;
    u_long    keepalivetime;
    u_long    keepaliveinterval;
}
```

The meaning of the structure fields *keepalivetime* and *keepaliveinterval* are identical to the Registry values discussed in the *SO_KEEPALIVE* option presented earlier in this chapter. Once a keepalive is set, you can query for the current keepalive values by calling *WSAIoctl* with the *SIO_KEEPALIVE_VALS* ioctl command and supplying a *tcp_keepalive* structure as the output buffer. This ioctl command is available on Windows 2000 only.

SIO_RCVALL

WSAIoctl
unsigned int
None
2+
Receives all packets on the network

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

Using this ioctl command with the value *TRUE* allows the given socket to receive all IP packets on the network. This requires that the socket handle be passed to *WSAIoctl* and that the socket be of address family *AF_INET* , socket type *SOCK_RAW* , and protocol *IPPROTO_IP* . Additionally, the socket must be bound to an explicit local interface. That is, you cannot bind to *INADDR_ANY* . Once the socket is bound and the ioctl is set, calls to *recv/WSARecv* return IP datagrams. Keep in mind that these are datagrams—you must supply a sufficiently large buffer. Because the total length field of the IP header is a 16-bit quantity, the maximum theoretical limit is 65,535 bytes; however, in practice the maximum transmission unit (MTU) of networks is much lower. Using this ioctl command requires Administrator privileges on the local machine. Additionally, this ioctl command is available on Windows 2000 only. A sample application on the companion CD, Rcvall.c, illustrates using this and the other two *SIO_RCVALL* ioctl commands.

SIO_RCVALL_MCAST

WSAIoctl
unsigned int
None
2+
Receives all multicast packets on the network

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command is similar to the *SIO_RCVALL* command described earlier. The same usage rules mentioned for *SIO_RCVALL* also apply here except that the socket passed to *WSAIoctl* should be created with the protocol equal to *IPPROTO_IGMP* . The one difference is that only multicast IP traffic is returned, as opposed to all IP packets. This means that only IP packets destined for addresses in the range 224.0.0.0 through 239.255.255.255 are returned. This ioctl command is available on Windows 2000 only.

SIO_RCVALL_IGMPMCAST

WSAIoctl
unsigned int
None
2+
Receives all IGMP packets on the network

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

Again, this ioctl command is the same as *SIO_RCVALL* , except that the socket passed into *WSAIoctl* should be created with the protocol equal to *IPPROTO_IGMP* . Setting this option returns only IGMP packets. See the *SIO_RCVALL* entry for instructions on how to use this option. This ioctl is available on Windows 2000 only.

SIO_ROUTING_INTERFACE_QUERY

Both
SOCKADDR
None
2+
Determines whether OOB data has been read

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command allows you to find the address of the local interface that should be used to send data to a remote machine. The address of the remote machine should be supplied in the form of a *SOCKADDR* structure as the *lpvInBuffer* parameter. Additionally, a sufficiently large buffer needs to be supplied as the *lpvOutBuffer* , which will contain an array of one or more *SOCKADDR* structures describing the local interface(s) that can be used. This command can be used for either unicast or multicast endpoints, and the interface returned from this call can be used in a subsequent call to *bind* .

The Windows 2000 plug-and-play capabilities are the motivation for having an ioctl like this. The user can insert or remove a PCMCIA network card, affecting which interfaces an application can use. A well-written application on Windows 2000 should take this into account.

Therefore, applications cannot rely on the information returned by *SIO_ROUTING_INTERFACE_QUERY* to be persistent. To handle this situation, you should also use the *SIO_ROUTING_INTERFACE_CHANGE* ioctl command, which notifies your application when the interfaces change. Once this occurs, call *SIO_ROUTING_INTERFACE_QUERY* once again to obtain the latest information.

SIO_ROUTING_INTERFACE_CHANGE

WSAIoctl
SOCKADDR
None
2+
Sends notification when an interface to an endpoint has changed

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

Using this ioctl command indicates that you want to be notified of any change in the local routing interface that is used to access the specified remote address. When you use this command, a *SOCKADDR* structure to the remote address in question is submitted in the input buffer and no data is returned upon successful completion. However, if the interface to that route changes in some way, the application will be notified, at which point the application can call *SIO_ROUTING_INTERFACE_QUERY* to determine which interface to use as a result.

There are several ways to make a call to this command. If the socket is blocking, the *WSAIoctl* call will not complete until some point at which the interface changes. If the socket is in nonblocking mode, the error *WSAEWOULDBLOCK* is returned. Then the application can wait for routing-change events through either

WSAEventSelect or *WSAAsyncSelect* , with the *FD_ROUTING_INTERFACE_CHANGE* flag set in the network event bitmask. Overlapped I/O can also be used to make the call. With this method, you supply an event handle in the *WSAOVERLAPPED* structure, which is signaled upon a routing change.

The address specified in the input *SOCKADDR* structure can be a specific address, or you can use the wildcard *INADDR_ANY* , indicating that you want to be notified of any routing changes. Note that because routing information remains fairly static, providers have the option of ignoring the information supplied by the application in the input buffer and simply sending a notification upon any interface change. As a result, it is probably a good idea to register for notification on any change and simply call *SIO_ROUTING_INTERFACE_QUERY* to see whether the change affects your application.

SIO_ADDRESS_LIST_QUERY

WSAIoctl
None
SOCKET_ADDRESS_LIST
2+
Returns a list of interfaces to which the socket can bind

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl is used to obtain a list of local transport addresses matching the socket's protocol family that the application can bind to. The output buffer is a *SOCKET_ADDRESS_LIST* structure, defined as

```
typedef struct _SOCKET_ADDRESS_LIST
{
INT            iAddressCount;
SOCKET_ADDRESS Address[1];
} SOCKET_ADDRESS_LIST, FAR * LPSOCKET_ADDRESS_LIST;

typedef struct _SOCKET_ADDRESS
{
    LPSOCKADDR lpSockaddr;
    INT        iSockaddrLength;
} SOCKET_ADDRESS, *PSOCKET_ADDRESS, FAR * LPSOCKET_ADDRESS;
```

The *iAddressCount* field returns the number of address structures in the list, while the *Address* field is an array of protocol family-specific addresses.

In Win32 plug-and-play environments, the number of valid addresses can change dynamically; therefore, applications cannot rely on the information returned from this ioctl command to remain constant. In order to take this into account, applications should first call *SIO_ADDRESS_LIST_QUERY* to obtain current interface information and then call *SIO_ADDRESS_LIST_CHANGE* to receive notification of future changes. If the address list changes, the application should again make a query.

If the supplied output buffer is not of sufficient size, *WSAIoctl* fails with *WSAEFAULT* , and the *lcbBytesReturned* parameter indicates the required buffer size.

SIO_ADDRESS_LIST_CHANGE

WSAIoctl
None
None
2+
Notifies when local interfaces change

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

An application can use this command to receive notification of changes in the list of local transport addresses of the given socket's protocol family to which the application can bind. No information is returned in the output

parameters upon successful completion of the call.

There are several ways to make a call to this command. If the socket is blocking, the *WSAIoctl* call will not complete until some point at which the interface changes. If the socket is in nonblocking mode, the error *WSAEWOULDBLOCK* is returned. Then the application can wait for routing-change events through either *WSAEventSelect* or *WSAAsyncSelect* with the *FD_ADDRESS_LIST_CHANGE* flag set in the network event bitmask. Additionally, overlapped I/O can be used to make the call. With this method, you supply an event handle in the *WSAOVERLAPPED* structure, which is signaled upon a routing change.

SIO_GET_INTERFACE_LIST

WSAIoctl
None
INTERFACE_INFO []
2+
Returns a list of local interfaces

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl is defined in Ws2tcpip.h. It is used to return information regarding each interface on the local machine. Nothing is required on input, but upon output, an array of *INTERFACE_INFO* structures is returned. The structures are defined as

```
typedef struct _INTERFACE_INFO
{
    u_long          iiFlags;           /* Interface flags   */
    sockaddr_gen    iiAddress;         /* Interface address */
    sockaddr_gen    iiBroadcastAddress; /* Broadcast address */
    sockaddr_gen    iiNetmask;         /* Network mask      */
} INTERFACE_INFO, FAR * LPINTERFACE_INFO;

#define IFF_UP            0x00000001 /* Interface is up                */
#define IFF_BROADCAST     0x00000002 /* Broadcast is supported         */
#define IFF_LOOPBACK      0x00000004 /* This is loopback interface     */
#define IFF_POINTTOPOINT  0x00000008 /* This is point-to-point interface*/
#define IFF_MULTICAST     0x00000010 /* Multicast is supported         */

typedef union sockaddr_gen
{
    struct sockaddr Address;
    struct sockaddr_in  AddressIn;
    struct sockaddr_in6 AddressIn6;
} sockaddr_gen;
```

The *iiFlags* member returns a bitmask of flags indicating whether the interface is up (*IFF_UP*) as well as whether broadcast (*IFF_BROADCAST*) or multicast (*IFF_MULTICAST*) is supported. It also indicates whether the interface is loopback (*IFF_LOOPBACK*) or point-to-point (*IFF_POINTTOPOINT*). The other three fields contain the address of the interface, the broadcast address, and the corresponding netmask.

Secure Socket Layer Ioctl Commands

Secure Socket Layer (SSL) commands are supported only in Windows CE. Currently Windows 95 and 98, Windows NT, and Windows 2000 do not provide an SSL-capable provider. As these are available only in Windows CE, the only version of Winsock currently supported for these options is version 1.

SO_SSL_GET_CAPABILITIES

WSAIoctl
None
DWORD
Returns the Winsock security provider's capabilities

Which Function?	*Input*	*Output*	*Description*

This command retrieves a set of flags describing the Windows Sockets security provider's capabilities. The output buffer must be a pointer to a *DWORD* bit field. At present, only the flag *SO_CAP_CLIENT* is defined.

SO_SSL_GET_FLAGS

WSAIoctl
None
DWORD
Returns *s* -channel-specific flags associated with socket

Which Function?	*Input*	*Output*	*Description*

This command retrieves *s* -channel_specific flags associated with a particular socket. The output buffer must be a pointer to a *DWORD* bit field. See *SO_SSL_SET_FLAGS* below for details of valid flags.

SO_SSL_SET_FLAGS

WSAIoctl
DWORD
None
Sets the socket's *s* -channel-specific flags

Which Function?	*Input*	*Output*	*Description*

Here the input buffer must be a pointer to a *DWORD* bit field. Currently only the *SSL_FLAG_DEFER_HANDSHAKE* flag, which allows the application to send and receive plain text data before switching to cipher text, is defined. This flag is required for setting up communication through proxy servers.

Normally the Windows Sockets security provider performs the secure handshake in the Windows Sockets *connect* function. However, if this flag is set, the handshake is deferred until the application issues the *SO_SSL_PERFORM_HANDSHAKE* control code. After the handshake, this flag is reset.

SO_SSL_GET_PROTOCOLS

WSAIoctl
None
SSLPROTOCOLS
Returns a list of protocols supported by the security provider

Which Function?	*Input*	*Output*	*Description*

This command retrieves a list of protocols that the provider currently supports on this socket. The output buffer must be a pointer to a *SSLPROTOCOLS* structure, as described here:

```
typedef struct _SSLPROTOCOL
{
    DWORD dwProtocol;
    DWORD dwVersion;
    DWORD dwFlags;
} SSLPROTOCOL, *LPSSLPROTOCOL;
typedef struct _SSLPROTOCOLS
{
    DWORD       dwCount;
```

```
    SSLPROTOCOL ProtocolList[1];
} SSLPROTOCOLS, FAR *LPSSLPROTOCOLS;
```

Valid protocols for the *dwProtocol* field include *SSL_PROTOCOL_SSL2* , *SSL_PROTOCOL_SSL3* , and *SSL_PROTOCOL_PCT1* .

SO_SSL_SET_PROTOCOLS

WSAIoctl
SSLPROTOCOLS
None
Sets a list of protocols that the underlying provider should support

Which Function?	*Input*	*Output*	*Description*

This ioctl command specifies a list of protocols that the provider is to support on this socket. The input buffer must be a pointer to the *SSLPROTOCOLS* structure described above.

SO_SSL_SET_VALIDATE_CERT_HOOK

WSAIoctl
SSLVALIDATECERTHOOK
None
Sets the validation function for accepting SSL certificates

Which Function?	*Input*	*Output*	*Description*

This ioctl command sets the pointer to the socket's certificate validation hook. It is used to specify the callback function invoked by the Windows Sockets security provider when a set of credentials is received from the remote party. The input buffer must be a pointer to the *SSLVALIDATECERTHOOK* structure, described as follows:

```
typedef struct
{
    SSLVALIDATECERTFUNC    HookFunc;
    LPVOID                 pvArg;
} SSLVALIDATECERTHOOK, *PSSLVALIDATECERTHOOK;
```

The *HookFunc* field is a pointer to a certificate validation callback function; *pvArg* is a pointer to application-specific data and can be used by the application for any purpose.

SO_SSL_PERFORM_HANDSHAKE

WSAIoctl
None
None
Initiates a secure handshake on a connected socket

Which Function?	*Input*	*Output*	*Description*

This ioctl command initiates the secure handshake sequence on a connected socket in which the *SSL_FLAG_DEFER_HANDSHAKE* flag has been set prior to the connection. Data buffers are not required, but the *SSL_FLAG_DEFER_HANDSHAKE* flag will be reset.

ATM Ioctl Commands

These ioctl commands are specific to the ATM protocol family. They are fairly basic, dealing mainly with obtaining the number of ATM devices and ATM addresses of the local interfaces. See Chapter 6 for more detailed information

about the addressing mechanisms for ATM.

SIO_GET_NUMBER_OF_ATM_DEVICES

WSAIoctl
None
DWORD
2+
Returns the number of ATM adapters

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command fills the output buffer pointed to by *lpvOutBuffer* with a *DWORD* containing the number of ATM devices in the system. Each specific device is identified by a unique ID, in the range 0 to the number returned by this ioctl command minus 1.

SIO_GET_ATM_ADDRESS

WSAIoctl
DWORD
ATM_ADDRESS
2+
Returns the ATM address for the given device

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command retrieves the local ATM address associated with the specified device. A device ID of type *DWORD* is specified in the input buffer for this ioctl command, and the output buffer pointed to by *lpvOutBuffer* will be filled with an *ATM_ADDRESS* structure containing a local ATM address suitable for use with *bind* .

SIO_ASSOCIATE_PVC

WSAIoctl
ATM_PVC_PARAMS
None
2+
Associates socket with a permanent virtual circuit

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command associates the socket with a permanent virtual circuit (PVC), as indicated in the input buffer, which contains the *ATM_PVC_PARAMS* structure. The socket should be of the *AF_ATM* address family. After successfully returning from this function, the application is able to start sending and receiving data as if the connection has been set up.

The *ATM_PVC_PARAMS* structure is defined as

```
typedef struct
{
    ATM_CONNECTION_ID   PvcConnectionId;
    QOS                 PvcQos;
} ATM_PVC_PARAMS;

typedef struct
{
    DWORD  DeviceNumber;
    DWORD  VPI;
    DWORD  VCI;
} ATM_CONNECTION_ID;
```

SIO_GET_ATM_CONNECTION_ID

Both
None
ATM_CONNECTION_ID
2+
Determines whether OOB data has been read

Which Function?	*Input*	*Output*	*Winsock Version*	*Description*

This ioctl command retrieves the ATM Connection ID associated with the socket. Upon successfully returning from this function, the output buffer pointed to by *lpvOutBuffer* is filled with an *ATM_CONNECTION_ID* structure containing the device number and VPI/VCI values, which are defined in the earlier entry for *SIO_ASSOCIATE_PVC* .

Conclusion

Such an enormous variety of socket options and ioctl commands might seem overwhelming at first, but they do allow applications to access protocol-specific characteristics, as well as offer you the ability to fine-tune an application. In some cases, an application must use one or more socket options or ioctls in order to operate, as in the case of AppleTalk or IrDA. Even so, an application will most likely use only a few options at a time. Of course, one of the more frustrating aspects of socket options and ioctls is the fact that not all options or ioctls are available on every Windows platform, causing trouble for those applications that are attempting to be cross platform-compatible.

Chapter 10

Registration and Name Resolution

This chapter covers the protocol-independent name registration and resolution model introduced by Winsock 2. The method introduced by Winsock 1 is now obsolete; therefore, we will not cover it. We will first give a bit of background on the importance and uses of name registration and resolution. Then we will move into the different types of name registration models available, followed by a description of the functions provided by Winsock 2 that can be used to resolve names. We will also cover how to register your own services for others to look up.

Background

Name registration is the process of associating a user-friendly name with a protocol-specific address. Host names and their IP addresses are a good example. Most people find it cumbersome to remember a workstation's address as, for example, "157.54.185.186." They would rather name their machines something easier to remember, such as "ajones1." In the case of IP, a service called Domain Name System (DNS) maps IP addresses to names. Other protocols offer ways of registering their specific addresses to friendlier names. Name spaces will be discussed in more detail in the next section.

Not only do you want to be able to register and resolve host names, you would also like the ability to map the address of your Winsock server so that clients can retrieve the address in order to connect to the server. For example, you might have a server running on machine 157.64.185.186 off port 5000. If the server runs only on that machine, you can always hardcode the server address in the client application. But what if you wanted a more dynamic server—one that can run on multiple machines, perhaps a distributed application with fault tolerance? If one server crashed or was too busy, another instance could be started somewhere else to service clients. In this case, finding out where the servers are possibly running can create headaches. Ideally, you want the ability to register your server—named "Fault Tolerant Distributed Server"—with multiple addresses. In addition, you want to be able to dynamically update the registered service and its addresses. This is what name registration and resolution is all about, and this chapter will address the facilities Winsock offers to accommodate distributed server registration and name resolution.

Name Space Models

Before we begin to explore the Winsock function, we need to introduce the various name space models to which most of the protocols adhere. A name space offers the capability to associate the protocol and its addressing attributes with a user-friendly name. Some of the more common name spaces are DNS for IP and the NetWare Directory Services (NDS) from Novell for IPX. These name spaces vary widely in their organization and implementation. Some of their properties are particularly important in understanding how to register and resolve names from Winsock.

There are three different types of name spaces: dynamic, static, and persistent. A dynamic name space allows you to register a service on the fly. This also means that clients can look up the service at run time. Typically, a dynamic name space relies on periodically broadcasting service information to signal that the service is continuously available. Examples of dynamic name spaces include Service Advertising Protocol (SAP)—used in NetWare environments—and AppleTalk's Name Binding Protocol (NBP) name space.

Static name spaces are the least flexible of the three types. Registering a service in a static name space requires that it be manually registered ahead of time. This means that there is no way to register a name with a static name space from Winsock—there is only a method of resolving names. DNS is an example of a static name space. For example, with DNS you manually enter IP addresses and host names into a file that the DNS service uses to handle resolution requests.

Persistent name spaces, like dynamic name spaces, allow services to register on the fly. Unlike dynamic name spaces, however, the persistent model maintains the registration information in nonvolatile storage, such as a file on a disk. Only when the service requests that it be removed will a persistent name space delete its entry. The advantage of a persistent name space is that it is flexible yet does not continually broadcast any kind of availability information. The drawback is that if a service is not well behaved (or is poorly written), it can go away without ever notifying the name space provider to remove its service entry, leading clients to believe incorrectly that the service is still available. NDS is an example of a persistent name space.

Enumerating Name Spaces

Now that you are acquainted with the various attributes of a name space, let's examine how to find out which name spaces are available on a machine. Most of the predefined name spaces are declared in the Nspapi.h header file. Each name space has an integer value assigned to it. Table 10-1 contains some of the more commonly supported name spaces available on Win32 platforms. The name spaces returned depend on which protocols are installed on the workstation. For example, unless IPX/SPX is installed on a workstation, the *NS_SAP* name space will not be returned.

Table 10-1. *Supported name spaces*

Name Space	*Value*	*Description*
NS_SAP	1	SAP name space; used on IPX networks
NS_NDS	2	NDS name space; also used on IPX networks
NS_DNS	11	DNS name space; most commonly found on TCP/IP networks and on the Internet
ND_NTDS	32	Windows NT domain space; protocol-independent name space found on Windows 2000

When you install IPX/SPX on a machine, the SAP name space is supported for queries only. If you want to register your own service, you also need to install the SAP Agent service. In some cases, the Client Services for NetWare are required to display local IPX interface addresses correctly. Without this service, the local addresses show up as all zeros. Additionally, you must add an NDS client to utilize the NDS name space. All of these protocols and services can be added from the Control Panel.

Winsock 2 provides a method of programmatically obtaining a list of the name spaces available on a system. This is accomplished by calling the function *WSAEnumNameSpaceProviders*, which is defined as

```
INT WSAEnumNameSpaceProviders (
    LPDWORD lpdwBufferLength,
    LPWSANAMESPACE_INFO lpnspBuffer
);
```

The first parameter is the size of the buffer submitted as *lpnspBuffer*, which is a sufficiently large array of *WSANAMESPACE_INFO* structures. If the function is called with an insufficiently large buffer, it fails, sets *lpdwBufferLength* to the required minimum size, and causes *WSAGetLastError* to return *WSAEFAULT*. The function returns the number of *WSANAMESPACE_INFO* structures returned, or *SOCKET_ERROR* upon any error.

The *WSANAMESPACE_INFO* structure describes an individual name space installed on the machine. This structure is defined as

```
typedef struct _WSANAMESPACE_INFO {
GUID NSProviderId;
DWORD dwNameSpace;
BOOL fActive;
DWORD dwVersion;
LPTSTR lpszIdentifier;
} WSANAMESPACE_INFO, *PWSANAMESPACE_INFO,
LPWSANAMESPACE_INFO;
```

There are actually two definitions for this structure—one is Unicode, and one is ANSI. The Winsock 2 header file type-defines the appropriate structure to *WSANAMESPACE_INFO* according to how you build your project. In actuality, all structures and Winsock 2 registration and name resolution functions have both ANSI and UNICODE versions. The first member of this structure, *NSProviderId*, is a globally unique identifier (GUID) that describes this particular name space. The *dwNameSpace* field is the name space's integer constant, such as *NS_DNS* or *NS_SAP*. The *fActive* member is a Boolean value, which if true indicates that the name space is available and ready to take queries; otherwise, the provider is inactive and unable to take queries that specifically reference this provider. The *dwVersion* field simply identifies the version of this provider. Finally the *lpszIdentifier* is a descriptive string identifier for this provider.

Registering a Service

The next step is to find out how to set up your own service and make it available and known to other machines on the network. This is known as registering an instance of your service with the name space provider so that it can either be advertised or queried by clients that want to communicate with it. Registering a service is actually a two-step process. The first step is to install a *service class* that describes the characteristics of your service.

It is important to distinguish between a service class and the actual service itself. For example, the service class describes which name spaces your service is to be registered with as well as certain characteristics about the service, such as whether it is connection-oriented or connectionless. The service class in no way describes how a client can establish a connection. Once the service class is registered, you register an actual instance of your service that references the correct service class to which it belongs. Once this occurs, a client can perform a query to find out where your service instance is running and therefore can attempt a communication.

Installing a Service Class

Before you register an instance of a service, you need to define the service class to which your service will belong. A service class defines what name spaces a service belonging to this class is registered with. The Winsock function that registers a service class is *WSAInstallServiceClass*, which is defined as

```
INT WSAInstallServiceClass (LPWSASERVICECLASSINFO lpServiceClassInfo);
```

The single parameter *lpServiceClassInfo* points to a *WSASERVICECLASSINFO* structure that defines the attributes of this class. The structure is defined as

```
typedef struct _WSAServiceClassInfo {
    LPGUID              lpServiceClassId;
    LPTSTR              lpszServiceClassName;
    DWORD               dwCount;
    LPWSANSCLASSINFO    lpClassInfos;
} WSASERVICECLASSINFO, *PWSASERVICECLASSINFO, LPWSASERVICECLASSINFO;
```

The first field is a GUID that uniquely identifies this particular service class. There are a couple ways to generate a GUID to use here. One way is to use the utility Uuidgen.exe and create a GUID for this service class. The problem with this method is that if you need to refer back to this GUID, you pretty much have to hardcode its value into a header file somewhere. This is where the second solution is useful. Within the header file Svcguid.h, several macros generate a GUID based on a simple attribute. For example, if you install a service class for SAP that will be used to advertise your IPX application, you can use the *SVCID_NETWARE* macro. The only parameter is the SAP ID number you assign to your "class" of applications. A number of SAP IDs are predefined in NetWare, such as 0x4 for file servers and 0x7 for a print server. Using this method, all you need is the easy-to-remember SAP ID to generate the GUID for the corresponding service class. Additionally, several macros exist that accept a port number as a parameter and return the GUID of the corresponding service. Take a look at the header file Svcguid.h, which contains other useful macros for the reverse operation—extracting the service port number from a GUID. Table 10-2 lists the most commonly used macros for generating GUIDs from simple protocol attributes such as port numbers or SAP IDs. The header file also contains constants for well-known port numbers for services such as FTP and Telnet.

Table 10-2. *Common service ID macros*

Macro	*Description*
SVCID_TCP(Port)	Generates a GUID from TCP port number
SVCID_DNS(RecordType)	Generates a GUID from a DNS record type
SVCID_UDP(Port)	Generates a GUID from UDP port number
SVCID_NETWARE(SapId)	Generates a GUID from SAP ID number

The second field of the *WSASERVICECLASSINFO* structure, *lpszServiceClassName*, is simply a string name for this particular service class. The last two fields are related. The *dwCount* field refers to the number of *WSANSCLASSINFO* structures passed in the *lpClassInfos* field. These structures define the name spaces and protocol characteristics that apply to the actual services that register under this service class. The structure is defined as

```
typedef struct _WSANSClassInfo {
    LPSTR   lpszName;
    DWORD   dwNameSpace;
    DWORD   dwValueType;
    DWORD   dwValueSize;
    LPVOID  lpValue;
}WSANSCLASSINFO, *PWSANSCLASSINFO, *LPWSANSCLASSINFO;
```

The *lpszName* field defines the attribute that the service class possesses. Table 10-3 lists the various attributes available. Every attribute listed has a value type of *REG_DWORD*.

Table 10-3. *Service types*

String Value	*Constant Define*	*Name Space*	*Description*
"SapId"	*SERVICE_TYPE_VALUE_SAPID*	*NS_SAP*	SAP ID
"ConnectionOriented"	*SERVICE_TYPE_VALUE_CONN*	Any	Indicates whether service is connection-oriented or connectionless
"TcpPort"	*SERVICE_TYPE_VALUE_TCPPORT*	*NS_DNS NS_NTDS*	TCP port
"UdpPort"	*SERVICE_TYPE_VALUE_UDPPORT*	*NS_NTDS NS_DNS*	UDP port

The *dwNameSpace* is the name space for which this attribute applies. Table 10-3 also lists the name spaces to which the various service types usually apply. The last three fields, *dwValueType*, *dwValueSize*, and *lpValue* all describe the actual value associated with the service type. The *dwValueType* field signifies the type of data associated with this entry and therefore can be one of the registry type values. For example, if the value is a *DWORD*, the value type is *REG_DWORD*. The next field, *dwValueSize*, is simply the size of the data passed as *lpValue,* which is a pointer to the data.

The following code example illustrates how to install a service class named "Widget Server Class."

```
WSASERVICECLASSINFO    sci;
WSANSCLASSINFO        aNameSpaceClassInfo[4];
DWORD            dwSapId = 200,
                dwUdpPort = 5150,
                dwZero = 0;
int             ret;

memset(&sci, 0, sizeof(sci));

SET_NETWARE_SVCID(&sci.lpServiceClassId, dwSapId);
sci.lpszServiceClassName = (LPSTR)"Widget Server Class";
sci.dwCount = 4;
sci.lpClassInfos = aNameSpaceClassInfo;

memset(aNameSpaceClassInfo, 0, sizeof(WSANSCLASSINFO) * 4);
// NTDS name space setup
aNameSpaceClassInfo[0].lpszName = SERVICE_TYPE_VALUE_CONN;
aNameSpaceClassInfo[0].dwNameSpace = NS_NTDS;
aNameSpaceClassInfo[0].dwValueType = REG_DWORD;
aNameSpaceClassInfo[0].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[0].lpValue = &dwZero;

aNameSpaceClassInfo[1].lpszName = SERVICE_TYPE_VALUE_UDPPORT;
aNameSpaceClassInfo[1].dwNameSpace = NS_NTDS;
aNameSpaceClassInfo[1].dwValueType = REG_DWORD;
aNameSpaceClassInfo[1].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[1].lpValue = &dwUdpPort;

// SAP name space setup
aNameSpaceClassInfo[2].lpszName = SERVICE_TYPE_VALUE_CONN;
aNameSpaceClassInfo[2].dwNameSpace = NS_SAP;
aNameSpaceClassInfo[2].dwValueType = REG_DWORD;
aNameSpaceClassInfo[2].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[2].lpValue = &dwZero;

aNameSpaceClassInfo[3].lpszName = SERVICE_TYPE_VALUE_SAPID;
aNameSpaceClassInfo[3].dwNameSpace = NS_SAP;
aNameSpaceClassInfo[3].dwValueType = REG_DWORD;
aNameSpaceClassInfo[3].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[3].lpValue = &dwSapId;

ret = WSAInstallServiceClass(&sci);
if (ret == SOCKET_ERROR)
{
    printf("WSAInstallServiceClass() failed %d\n", WSAGetLastError());
}
```

The first noticeable thing this example does is to pick a GUID that this class will be registered under. The services you are designing all belong to the class "Widget Server Class", and this service class describes the general attributes belonging to an instance of the service. In this example, we chose to register this class with the NetWare SAP ID of 200. This is only for convenience. We could have picked an arbitrary GUID or even the GUID

based on the UDP port number. Additionally, the service can use the UDP protocol, in which case the clients are listening on port 5150.

The next step of note is setting the *dwCount* field of the *WSASERVICECLASSINFO* to 4. In this example, you will register this service class with both the SAP name space (*NS_SAP*) and the Windows NT domain space (*NS_NTDS*). The odd thing you'll notice is that we use four *WSANSCLASSINFO* structures, even though we are registering the service class with only two name spaces. This is because we define two attributes for each name space and each attribute requires a separate *WSANSCLASSINFO* structure. For each name space, we define whether the service will be connection-oriented. In this example, the name space is connectionless, as we set the value for *SERVICE_TYPE_VALUE_CONN* to be a Boolean 0. For the Windows NT domain space, we also set the UDP port number this service normally runs under by using the service type *SERVICE_TYPE_VALUE_UDPPORT*. For the SAP name space, we set the SAP ID of our service with service type *SERVICE_TYPE_VALUE_SAPID*.

For every *WSANSCLASSINFO* entry, you must set the name space identifier to which this service type applies, as well as the type and size of the value. Table 10-3 contains the types required for the service types, which all turn out to be *DWORD* in the example. The last step is simply to call *WSAInstallServiceClass* and pass the *WSASERVICECLASSINFO* structure as the parameter. If *WSAInstallServiceClass* is successful, the function returns 0; otherwise, it returns *SOCKET_ERROR*. If *WSASERVICECLASSINFO* is invalid or improperly formed, *WSAGetLastError* returns *WSAEINVAL*. If the service class already exists, *WSAGetLastError* returns *WSAEALREADY*. In this case, a service class can be removed by calling *WSARemoveServiceClass*, which is declared as

```
INT WSARemoveServiceClass( LPGUID lpServiceClassId );
```

This function's only parameter is a pointer to the GUID that defines the given service class.

Service Registration

Once you have a service class installed that describes the general attributes of your service, you can register an instance of your service so that it is available for lookup by other clients on remote machines. The Winsock function to register an instance of a service is *WSASetService*.

```
INT WSASetService (
    LPWSAQUERYSET lpqsRegInfo,
    WSAESETSERVICEOP essOperation,
    DWORD dwControlFlags
);
```

The first parameter, *lpqsRegInfo*, is a pointer to a *WSAQUERYSET* structure that defines the particular service. We'll discuss what goes in this structure shortly. The *essOperation* parameter specifies the action to take place, such as registration or deregistration. Table 10-4 describes the three valid flags.

The third parameter, *dwControlFlags*, is either 0 or the flag *SERVICE_MULTIPLE*. This flag is used if multiple addresses will be registered under the given service instance. For example, say you have a service that you want to run on five machines. The *WSAQUERYSET* structure passed into *WSASetService* would reference five *CSADDR_INFO* structures, each describing the location of one instance of the service. This requires the *SERVICE_MULTIPLE* flag to be set. Additionally, at some later point you can deregister a single instance of the service by using the *RNRSERVICE_DELETE* service flag. Table 10-5 gives the possible combinations of the operation and control flags and describes the result of the command, depending on whether the service already exists.

Table 10-4. *Set service flags*

Operation Flag	*Meaning*
RNRSERVICE_REGISTER	Register the service. For dynamic name providers, this means to begin actively advertising the service. For persistent name providers, this means updating the database. For static name providers, this does nothing.
RNRSERVICE_DEREGISTER	Remove the entire service from the registry. For dynamic name providers, this means to stop advertising the service. For persistent name providers, this means removing the service from the database. For static name providers, this does nothing.
RNRSERVICE_DELETE	Remove only the given instance of the service from the name space. A service might be registered that contains multiple instances (using the *SERVICE_MULTIPLE* flag upon registration), and this command removes only the given instance of the service (as defined by a *CSADDR_INFO* structure). Again, this applies only to dynamic and persistent name providers.

Table 10-5. WSASetService *flag combinations*

RNRSERVICE_REGISTER		
Flags	*Meaning*	
	If the Service Already Exists	*If the Service Does Not Exist*
none	Overwrite the existing service instance.	Add a new service entry on the given address.
SERVICE_MULTIPLE	Update the service instance by adding the new addresses.	Add a new service entry on the given addresses.
RNRSERVICE_DEREGISTER		
Flags	*Meaning*	
	If the Service Already Exists	*If the Service Does Not Exist*
none	Remove all instances of the service, but do not remove the service. (Basically, *WSAQUERYSET* remains, but the number of *CSADDR_INFO* structures is 0.)	This is an error, and *WSASERVICE_NOT_FOUND* is returned.
SERVICE_MULTIPLE	Update the service by removing the given addresses. The service remains registered, even if no addresses remain.	This is an error, and *WSASERVICE_NOT_FOUND* is returned.
RNRSERVICE_DELETE		
Flags	*Meaning*	
	If the Service Already Exists	*If the Service Does Not Exist*
none	The service is removed completely from the name space.	This is an error, and *WSASERVICE_NOT_FOUND* is returned.
SERVICE_MULTIPLE	Update the service by removing the given addresses. If no addresses remain, the service is completely removed from the name space.	This is an error, and *WSASERVICE_NOT_FOUND* is returned.

Now that you have an understanding of what *WSASetService* does, let's take a look at the *WSAQUERYSET* structure that needs to be filled out and passed into the function. This structure is defined as

```
typedef struct _WSAQuerySetW {
    DWORD           dwSize;
    LPTSTR          lpszServiceInstanceName;
    LPGUID          lpServiceClassId;
    LPWSAVERSION    lpVersion;
    LPTSTR          lpszComment;
    DWORD           dwNameSpace;
    LPGUID          lpNSProviderId;
    LPTSTR          lpszContext;
    DWORD           dwNumberOfProtocols;
    LPAFPROTOCOLS   lpafpProtocols;
    LPTSTR          lpszQueryString;
    DWORD           dwNumberOfCsAddrs;
    LPCSADDR_INFO   lpcsaBuffer;
    DWORD           dwOutputFlags;
    LPBLOB          lpBlob;
} WSAQUERYSETW, *PWSAQUERYSETW, *LPWSAQUERYSETW;
```

The *dwSize* field should be set to the size of the *WSAQUERYSET* structure. The *lpszServiceInstanceName* field contains a string identifier naming this instance of the server. The *lpServiceClassId* field is the GUID for the service class to which this service instance belongs. The *lpVersion* field is optional. You can use it to supply version information that could be useful when a client queries for a service. The *lpszComment* field is also optional. You can specify any kind of comment string here. The *dwNameSpace* field specifies the name spaces to register your service with. If you're using only a single name space, use only that value; otherwise, use *NS_ALL*. It is possible to reference a custom name space provider. (Writing your own name space is discussed in Chapter 14.) For a custom name space provider, the *dwNameSpace* field is set to 0 and *lpNSProviderId* specifies the GUID representing the custom provider. The *lpszContext* field specifies the starting point of the query in a hierarchical name space such as NDS.

The *dwNumberOfProtocols* and *lpafpProtocols* fields are optional parameters used to narrow the search to return only the supplied protocols. The *dwNumberOfProtocols* field references the number of *AFPROTOCOLS* structures contained in the *lpafpProtocols* array. The structure is defined as

```
typedef struct _AFPROTOCOLS {
    INT iAddressFamily;
    INT iProtocol;
} AFPROTOCOLS, *PAFPROTOCOLS, *LPAFPROTOCOLS;
```

The first field, *iAddressFamily*, is the address family constant, such as *AF_INET* or *AF_IPX*. The second field, *iProtocol*, is the protocol from the given address family, such as *IPPROTO_TCP* or *NSPROTO_IPX*.

The next field in the *WSAQUERYSET* structure, *lpszQueryString*, is optional and used only by name spaces supporting enriched Structured Query Language (SQL) queries such as Whois++. This parameter is used to specify that string.

The next two fields are the most important when registering a service. The *dwNumberOfCsAddrs* field simply provides the number of *CSADDR_INFO* structures passed in *lpcsaBuffer*. The *CSADDR_INFO* structure defines the address family and the actual address at which the service is located. If multiple structures are present, multiple instances of the service are available. The structure is defined as

```
typedef struct _CSADDR_INFO {
    SOCKET_ADDRESS LocalAddr;
    SOCKET_ADDRESS RemoteAddr;
    INT            iSocketType;
    INT            iProtocol;
} CSADDR_INFO;

typedef struct _SOCKET_ADDRESS {
    LPSOCKADDR lpSockaddr;
    INT        iSockaddrLength;
} SOCKET_ADDRESS, *PSOCKET_ADDRESS, FAR * LPSOCKET_ADDRESS;
```

Additionally, the definition of *SOCKET_ADDRESS* is included. When registering a service, you can specify the local and remote addresses. The local address field (*LocalAddr*) is used to specify the address that an instance of this service should bind to, while the remote address field (*RemoteAddr*) is the address a client should use in a *connect* or a *sendto* call. The other two fields, *iSocketType* and *iProtocol*, specify the socket type (for example, *SOCK_STREAM*, *SOCK_DGRAM*) and the protocol family (for example, *AF_INET*, *AF_IPX*) for the given addresses.

The last two fields of the *WSAQUERYSET* structure are *dwOutputFlags* and *lpBlob*. These two fields are generally not needed for service registration; they are more useful when querying for a service instance (covered in the next section). Only the name space provider can return a *BLOB* structure. That is, when registering a service you cannot add your own *BLOB* structure to be returned in client queries.

Table 10-6 lists the fields of the *WSAQUERYSET* structure and identifies which are required or optional depending on whether a query or a registration is being performed.

Service registration example

In this section, we'll show you how to register your own service under both the SAP and NTDS name spaces. The Windows NT domain space is quite powerful, which is why we want to include it in our example. However, be aware of the following features. First the Windows NT domain space requires Windows 2000 since it is based on the Active Directory. This also means that the Windows 2000 workstation on which you hope to register and/or look up services must have a machine account in that domain in order to access the Active Directory. The other feature to note is that the Windows NT domain space is capable of registering socket addresses from any protocol family. This means that your IP and IPX services can all be registered in the same name space. It also means that there is a dynamic way of adding and removing IPbased services. Figure 10-1 illustrates the basic steps required to register an instance of a service. For the sake of simplicity, no error checking is performed.

Table 10-6. WSAQUERYSET *fields*

Field	*Query*	*Registration*
dwSize	Required	Required
lpszServiceInstanceName	String or "*" required	Required
lpServiceClassId	Required	Required
lpVersion	Optional	Optional
lpszComment	Ignored	Optional
dwNameSpace *lpNSProviderId*	One of these two fields must be specified	One of these two fields must be specified
lpszContext	Optional	Optional
dwNumberOfProtocols	Zero or more	Zero or more
lpafpProtocols	Optional	Optional
lpszQueryString	Optional	Ignored
dwNumberOfCsAddrs	Ignored	Required
lpcsaBuffer	Ignored	Required
dwOutputFlags	Ignored	Optional
lpBlob	Ignored, can be returned by the query	Ignored

Figure 10-1. *The WSASetService example*

```
SOCKET              socks[2];
WSAQUERYSET         qs;
CSADDR_INFO         lpCSAddr[2];
SOCKADDR_IN         sa_in;
SOCKADDR_IPX        sa_ipx;
IPX_ADDRESS_DATA    ipx_data;
GUID                guid = SVCID_NETWARE(200);
int                 ret, cb;

memset(&qs, 0, sizeof(WSAQUERYSET));
qs.dwSize = sizeof(WSAQUERYSET);
qs.lpszServiceInstanceName = (LPSTR)"Widget Server";
qs.lpServiceClassId = &guid;
qs.dwNameSpace = NS_ALL;
qs.lpNSProviderId = NULL;
qs.lpcsaBuffer = lpCSAddr;
qs.lpBlob = NULL;
//
// Set the IP address of our service
//
memset(&sa_in, 0, sizeof(sa_in));
sa_in.sin_family = AF_INET;
sa_in.sin_addr.s_addr =  htonl(INADDR_ANY);
sa_in.sin_port = 5150;
```

```
socks[0] = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
ret = bind(socks[0], (SOCKADDR *)&sa_in, sizeof(sa_in));

cb = sizeof(sa_in);
getsockname(socks[0], (SOCKADDR *)&sa_in, &cb);

lpCSAddr[0].iSocketType = SOCK_DGRAM;
lpCSAddr[0].iProtocol = IPPROTO_UDP;
lpCSAddr[0].LocalAddr.lpSockaddr = (SOCKADDR *)&sa_in;
lpCSAddr[0].LocalAddr.iSockaddrLength = sizeof(sa_in);
lpCSAddr[0].RemoteAddr.lpSockaddr = (SOCKADDR *)&sa_in;
lpCSAddr[0].RemoteAddr.iSockaddrLength = sizeof(sa_in);
//
// Set up the IPX address for our service
//
memset(sa_ipx.sa_netnum, 0, sizeof(sa_ipx.sa_netnum));
memset(sa_ipx.sa_nodenum, 0, sizeof(sa_ipx.sa_nodenum));
sa_ipx.sa_family = AF_IPX;
sa_ipx.sa_socket = 0;

socks[1] = socket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX);

ret = bind(socks[1], (SOCKADDR *)&sa_ipx, sizeof(sa_ipx));

cb = sizeof(IPX_ADDRESS_DATA);
memset (&ipx_data, 0, cb);
ipx_data.adapternum = 0;

ret = getsockopt(socks[1], NSPROTO_IPX, IPX_ADDRESS,
    (char *)&ipx_data, &cb);

cb = sizeof(SOCKADDR_IPX);
getsockname(socks[1], (SOCKADDR *)sa_ipx, &cb);

memcpy(sa_ipx.sa_netnum, ipx_data.netnum, sizeof(sa_ipx.sa_netnum));
memcpy(sa_ipx.sa_nodenum, ipx_data.nodenum, sizeof(sa_ipx.sa_nodenum));

lpCSAddr[1].iSocketType = SOCK_DGRAM;
lpCSAddr[1].iProtocol = NSPROTO_IPX;
lpCSAddr[1].LocalAddr.lpSockaddr = (struct sockaddr *)&sa_ipx;
lpCSAddr[1].LocalAddr.iSockaddrLength = sizeof(sa_ipx);
lpCSAddr[1].RemoteAddr.lpSockaddr = (struct sockaddr *)&sa_ipx;
lpCSAddr[1].RemoteAddr.iSockaddrLength = sizeof(sa_ipx);

qs.dwNumberOfCsAddrs = 2;

ret = WSASetService(&qs, RNRSERVICE_REGISTER, 0L);
```

The example code in Figure 10-1 illustrates how to set an instance of a service so that a client of that service can find out the address that it requires to communicate with the service. The first order of business is to initialize the *WSAQUERYSET* structure. We also need to give a name to the instance of our service. In this case, we simply call it "Widget Server." The other critical step is to use the same GUID we used to register our service class. Whenever you register an instance of a service, that service must belong to a service class. In this case, we use the "Widget

Service Class" (defined in the previous section), whose GUID is *SVCID_NETWARE(200)*. The next step is to set the name spaces in which we are interested. Since our service runs over both IPX and UDP, we specify *NS_ALL*. Because we're specifying a preexisting name space, *lpNSProviderId* must be set to *NULL*.

The next step is to set up the *SOCKADDR* structures within the *CSADDR_INFO* array that *WSASetService* passes as the *lpcsaBuffer* field of the *WSAQUERYSET* structure. You'll notice that in our example we actually create the sockets and bind them to a local address before we set up the *SOCKADDR* structure. This is because we need to find the exact local address to which clients need to connect. For example, when creating our UDP socket for the server, we bind to *INADDR_ANY*, which doesn't give us the actual IP address until we call *getsockname*. Using the information returned from *getsockname*, we can build a *SOCKADDR_IN* structure. Within the *CSADDR_INFO* structure, we set the socket type and the protocol. The other two fields are the local and remote address information. The local address is the address that a server should bind to, while the remote address is the address that a client should use to connect to the service.

After setting up the *SOCKADDR_IN* structure for our UDP-based server, we set up the IPX-based service. In Chapter 6, you saw that servers should bind to the internal network number by setting the network and node number to 0. Again, this doesn't give you the address that clients need, so call the socket option *IPX_ADDRESS* to obtain the actual address. In filling the *CSADDR_INFO* structure for IPX, use *SOCK_DGRAM* and *NSPROTO_IPX* for the socket type and the protocol, respectively. The last step is to set the *dwNumberOfCsAddrs* field in the *WSAQUERYSET* structure to 2, as there are two addresses—UDP and IPX—that clients can use to establish a connection. Finally, call *WSASetService* with our *WSAQUERYSET* structure, the *RNRSERVICE_REGISTER* flag, and no control flags. You do not specify the *SERVICE_MULTIPLE* control flag so that if you choose to deregister our service, all instances of the service (both the IPX and UDP addresses) will be deregistered.

There is one consideration that the above example does not take into account: multihomed machines. If you create a UDP-based server that binds to *INADDR_ANY* on a multihomed machine, the client can connect to the server on any of the available interfaces. In the case of IP, *getsockname* is not sufficient; you must obtain all local IP interfaces. There are a number of methods of obtaining this information, depending on the platform you are on. One method common to all platforms is calling *gethostbyname* to return a list of IP addresses for our name. Under Winsock 2, you can also call the ioctl command *SIO_GET_INTERFACE_LIST*. For Windows 2000, the ioctl *SIO_ADDRESS_LIST_QUERY* is available. Finally, the IP helper functions discussed in Appendix B can be used as well. Simple TCP/IP name resolution and *gethostbyname* are presented in Chapter 6, while ioctl commands are found in Chapter 9. In addition, Rnrcs.c (on the companion CD) is a full-fledged example that addresses multihomed machines.

Querying a Service

Now that you know how to register a service within a name space, you'll take a look at how a client can query the name space for a given service so that the client can obtain information about the service for communication purposes. Name resolution is quite a bit simpler than service registration, even though name resolution uses three functions for querying: *WSALookupServiceBegin* , *WSALookupServiceNext* , and *WSALookupServiceEnd.*

The first step when performing a query is to call *WSALookupServiceBegin* , which initiates the query by setting up the constraints within which the query will act. The function prototype is as follows:

```
INT WSALookupServiceBegin (
    LPWSAQUERYSET lpqsRestrictions,
    DWORD dwControlFlags,
    LPHANDLE lphLookup
);
```

The first parameter is a *WSAQUERYSET* structure that places constraints on the query, such as limiting the name spaces to query. The second parameter, *dwControlFlags* , determines the depth of the search. Table 10-7 contains the various possible flags and their meanings. These flags affect how the query behaves as well as which data is returned from the query. The last parameter is of type *HANDLE* and is initialized upon function return. The return value is 0 on success; otherwise, *SOCKET_ERROR* is returned. If one or more parameters are invalid, *WSAGetLastError* returns *WSAEINVAL* . If the name is found in the name space but no data matches the given restrictions, the error is *WSANO_DATA* . If the given service does not exist, *WSASERVICE_NOT_FOUND* is the error.

Table 10-7. *Control flags*

LUP_DEEP
In hierarchical name spaces, query deep as opposed to the first level.
LUP_CONTAINERS
Retrieve container objects only. This flag pertains to hierarchical name spaces only.
LUP_NOCONTAINERS
Do not return any containers. This flag pertains to hierarchical name spaces only.
LUP_FLUSHCACHE
Ignore cached information, and query the name space directly. Note that not all name providers cache queries.
LUP_FLUSHPREVIOUS
Instruct the name provider to discard the information set previously returned. This flag is typically used after *WSALookupServiceNext* returns *WSA_NOT_ENOUGH_MEMORY* . The information that was too big for the supplied buffer is discarded, and the next information set is to be retrieved.
LUP_NEAREST
Retrieve the results in order of distance. Note that it is up to the name provider to calculate this distance metric, as there is no specific provision for this information when a service is registered. Name providers are not required to support this concept.
LUP_RES_SERVICE
Specifies that the local address be returned in the *CSADDR_INFO* structure.
LUP_RETURN_ADDR
Retrieve the addresses as *lpcsaBuffer.*
LUP_RETURN_ALIASES
Retrieve only alias information. Each alias will be returned in successive calls to *WSALookupServiceNext* and will have the *RESULT_IS_ALIAS* flag set.
LUP_RETURN_ALL
Retrieve all available information.
LUP_RETURN_BLOB
Retrieve the private data as *lpBlob.*
LUP_RETURN_COMMENT
Retrieve the comment as *lpszComment.*
LUP_RETURN_NAME
Retrieve the name as *lpszServiceInstanceName.*
LUP_RETURN_TYPE
Retrieve the type as *lpServiceClassId.*
LUP_RETURN_VERSION
Retrieve the version as *lpVersion* .

Flag	Meaning

When you make a call to *WSALookupServiceBegin* , a handle for the query is returned that you pass to *WSALookupServiceNext* , which returns the data to you. The function is defined as

```
INT WSALookupServiceNext (
    HANDLE hLookup,
    DWORD dwControlFlags,
    LPDWORD lpdwBufferLength,
    LPWSAQUERYSET lpqsResults
);
```

The handle *hLookup* is returned from *WSALookupServiceBegin* . The *dwControlFlags* parameter has the same meaning as in *WSALookupServiceBegin* except that only *LUP_FLUSHPREVIOUS* is supported. The parameter *lpdwBufferLength* is the length of the buffer passed as *lpqsResults* . Because the *WSAQUERYSET* structure could contain binary large object (BLOB) data, it is often required that you pass a buffer larger than the structure itself. If the buffer size is insufficient for the data to be returned, the function call fails with *WSA_NOT_ENOUGH_MEMORY* .

Once you have initiated the query with *WSALookupServiceBegin* , call *WSALookupServiceNext* until the error *WSA_E_NO_MORE* (10110) is generated. One note of caution: in earlier implementations of Winsock, the error code for no more data is *WSAENOMORE* (10102), so robust code should check for both error codes. Once all the data has been returned or you have finished querying, call *WSALookupServiceEnd* with the *HANDLE* variable used in the queries. The function is

```
INT WSALookupServiceEnd ( HANDLE hLookup );
```

Forming a Query

Let's look at how you can query the service you registered in the previous section. The first thing to do is set up a *WSAQUERYSET* structure that defines the query. Look at the following code:

```
WSAQUERYSET    qs;
GUID        guid = SVCID_NETWARE(200);
AFPROTOCOLS      afp[2] = {{AF_IPX,  NSPROTO_IPX}, {AF_INET, IPPROTO_UDP}};
HANDLE        hLookup;
int        ret;

memset(&qs, 0, sizeof(qs));
qs.dwSize = sizeof (WSAQUERYSET);
qs.lpszServiceInstanceName = "Widget Server";
qs.lpServiceClassId = &guid;
qs.dwNameSpace = NS_ALL;
qs.dwNumberOfProtocols = 2;
qs.lpafpProtocols = afp;

ret = WSALookupServiceBegin(&qs, LUP_RETURN_ADDR | LUP_RETURN_NAME,
         &hLookup);
if (ret == SOCKET_ERROR)
```

```
    // Error
```

Remember that all service lookups are based on the service class GUID that the service you are searching for is based on. The variable *guid* is set to the service class ID of our server. You first initialize *qs* to 0 and set the *dwSize* field to the size of the structure. The next step is to give the name of the service you are searching for. The service name can be the exact name of the server, or you can specify a wildcard (*) that will return all services of the given service class GUID. Next you tell the query to search all name spaces by using the *NS_ALL* constant. Last you set up the protocols that our client is capable of connecting with, which are IPX and UDP/IP. This is done by using an array of two *AFPROTOCOLS* structures.

Now you are ready to begin the query, so you must call *WSALookupServiceBegin* . The first parameter is our *WSAQUERYSET* structure, while the next parameters are flags defining which data should be returned if a matching service is found. Here you specify that you want addressing information and the service name by logically ORing the two flags *LUP_RETURN_ADDR* and *LUP_RETURN_NAME* . The flag *LUP_RETURN_NAME* is necessary only if you're specifying the wildcard (*) for the service name; otherwise, you already know the name of the service. The last parameter is a *HANDLE* variable that identifies this particular query. It will be initialized upon successful return.

Once the query is successfully opened, you call *WSALookupServiceNext* until *WSA_E_NO_MORE* is returned. Each successful call returns information about a service that matches our criteria. Here is what the code looks like for this step:

```
char           buff[sizeof(WSAQUERYSET) + 2000];
DWORD          dwLength, dwErr;
WSAQUERYSET   *pqs = NULL;
SOCKADDR      *addr;
int            i;

pqs = (WSAQUERYSET *)buff;
dwLength = sizeof(WSAQUERYSET) + 2000;
while (1)
{
    ret = WSALookupServiceNext(hLookup, 0, &dwLength, pqs);
    if (ret == SOCKET_ERROR)
    {
        if ((dwErr = WSAGetLastError()) == WSAEFAULT)
        {
            printf("Buffer too small; required size is: %d\n", dwLength);
            break;
        }
        else if ((dwErr == WSA_E_NO_MORE) || (dwErr = WSAENOMORE))
            break;
        else
        {
            printf("Failed with error: %d\n", dwErr);
            break;
        }
    }
    for (i = 0; i < pqs->dwNumberOfCsAddrs; i++)
    {
        addr = (SOCKADDR *)pqs->lpcsaBuffer[i].RemoteAddr.lpSockaddr;
        if (addr->sa_family == AF_INET)
        {
            SOCKADDR_IN *ipaddr = (SOCKADDR_IN *)addr;
            printf("IP address:port = %s:%d\n", inet_ntoa(addr->sin_addr),
```

```
            addr->sin_port);
        }
        else if (addr->sa_family == AF_IPX)
        {
            SOCKADDR_IPX *ipxaddr = (SOCKADDR_IPX *)addr;
            printf("%02X%02X%02X%02X.%02X%02X%02X%02X%02X%02X:%04X",
                (unsigned char)ipxaddr->sa_netnum[0],
                (unsigned char)ipxaddr->sa_netnum[1],
                (unsigned char)ipxaddr->sa_netnum[2],
                (unsigned char)ipxaddr->sa_netnum[3],
                (unsigned char)ipxaddr->sa_nodenum[0],
                (unsigned char)ipxaddr->sa_nodenum[1],
                (unsigned char)ipxaddr->sa_nodenum[2],
                (unsigned char)ipxaddr->sa_nodenum[3],
                (unsigned char)ipxaddr->sa_nodenum[4],
                (unsigned char)ipxaddr->sa_nodenum[5],
                ntohs(ipxaddr->sa_socket));
        }
    }
}
WSALookupServiceEnd(hLookup);
```

The example code is straightforward, although a bit simplified. Calling *WSALookupServiceNext* requires only a valid handle to a query, the length of the return buffer, and the return buffer itself. You don't need to specify any control flags, as the only valid flag for this function is *LUP_FLUSHPREVIOUS* . If our supplied buffer is too small and this flag is set, the results from this call are discarded. However, in this example we don't use *LUP_FLUSHPREVIOUS* , and if our buffer is too small, the *WSAEFAULT* error is generated. If this occurs, *lpdwBufferLength* is set to the required size. Our example uses a fixed-size buffer equal to the size of the *WSAQUERYSET* structure plus 2000 bytes. Since all you are asking for are service names and addresses, this should be a sufficient size. Of course, in production code your applications should be prepared to handle the *WSAEFAULT* error.

Once you successfully call *WSALookupServiceNext* , the buffer is filled with a *WSAQUERYSET* structure containing the results. In our query, you asked for names and addresses; the fields of *WSAQUERYSET* that are of most interest are *lpszServiceInstanceName* and *lpcsaBuffer* . The former contains the name of the service, while the latter is an array of *CSADDR_INFO* structures that contains addressing information for the service. The parameter *dwNumberOfCsAddrs* tells us exactly how many addresses have been returned. In the example code, all we do is simply print out the addresses. Check only for IPX and IP addresses to print because those are the only address families you requested when you opened the query.

If our query used a wildcard (*) for the service name, each call to *WSALookupServiceNext* would return a particular instance of that service running somewhere on the network—provided, of course, that multiple instances are actually registered and running. Once all instances of the service have been returned, the error *WSA_E_NO_MORE* is generated, and you break out of the loop. The last thing you do is call *WSALookupServiceEnd* on the query handle. This releases any resources allocated for the query.

Querying DNS

Earlier we mentioned that the DNS namespace is static, which means you cannot dynamically register your service; however, you can still use the Winsock name resolution functions to perform a DNS query. Performing a DNS query is actually a bit more complicated than performing a normal query for a service that you have registered because the DNS name space provider returns the query information in the form of a BLOB. Why does it do this? Remember from the Chapter 6 discussion of *gethostbyname* that a name lookup returns a *HOSTENT* structure that contains not only IP addresses but also aliases. That information doesn't quite fit into the fields of the *WSAQUERYSET* structure.

The tricky thing about BLOB data is that the format of the data is not well documented, which makes directly querying DNS challenging. First let's take a look at how to open the query. The file Dnsquery.c on the companion CD contains the entire example code for querying DNS directly; however, we'll take a look at it piece by piece. The

following code illustrates initializing the DNS query:

```
WSAQUERYSET  qs;
AFPROTOCOLS  afp [2] = {{AF_INET, IPPROTO_UDP},{AF_INET, IPPROTO_TCP}};
GUID         hostnameguid = SVCID_INET_HOSTADDRBYNAME;
DWORD        dwLength = sizeof(WSAQUERYSET) + sizeof(HOSTENT) + 2048;
HANDLE       hQuery;

qs = (WSAQUERYSET *)buff;
memset(&qs, 0, sizeof(qs));
qs.dwSize = sizeof(WSAQUERYSET);
qs.lpszServiceInstanceName = argv[1];
qs.lpServiceClassId = &hostnameguid;
qs.dwNameSpace = NS_DNS;
qs.dwNumberOfProtocols = 2;
qs.lpafProtocols = afp;

ret = WSALookupServiceBegin(&qs, LUP_RETURN_NAME | LUP_RETURN_BLOB,
    &hQuery);
if (ret == SOCKET_ERROR)
    // Error
```

Setting up the query is quite similar to our previous example. The most noticeable change is that we use the predefined GUID *SVCID_INET_HOSTADDRBYNAME* . This is the GUID that identifies host name queries. The *lpszServiceInstanceName* is the host name that we want to resolve. Because we are resolving host names through DNS, we need to specify only *NS_DNS* for *dwNameSpace* . Finally, *lpafProtocols* is set to an array of two *AFPROTOCOLS* structures, which defines the TCP/IP and UDP/IP protocols as those that our query is interested in.

Once you establish the query, you can call *WSALookupServiceNext* to return data.

```
char         buff[sizeof(WSAQUERYSET) + sizeof(HOSTENT) + 2048)];
DWORD        dwLength = sizeof(WSAQUERYSET) + sizeof(HOSTENT) + 2048;
WSAQUERYSET *pqs;
HOSTENT     *hostent;

pqs = (WSAQUERYSET *)buff;
pqs->dwSize = sizeof(WSAQUERYSET);
ret = WSALookupServiceNext(hQuery, 0, &dwLength, pqs);
if (ret == SOCKET_ERROR)
    // Error
WSALookupServiceEnd(hQuery);

hostent = pqs->lpBlob->pBlobData;
```

Because a DNS name space provider returns the host information in the form of a BLOB, you need to supply a sufficiently large buffer. That is why you use a buffer equal in size to a *WSAQUERYSET* structure, plus a *HOSTENT* structure, plus 2048 bytes for good measure. Again, if this were insufficient, the call would fail with *WSAEFAULT* . In a DNS query, all the host information is returned within the *HOSTENT* structure, even if a host name is associated with multiple IP addresses. That is why you don't need to call *WSALookupServiceNext* multiple times.

Now comes the tricky part—decoding the *BLOB* structure returned by the query. From Chapter 6 , you know the *HOSTENT* structure is defined as

```
typedef struct hostent {
char FAR * h_name;
char FAR * FAR * h_aliases;
short h_addrtype;
short h_length;
char FAR * FAR * h_addr_list;
} HOSTENT;
```

When the *HOSTENT* structure is returned as the BLOB data, the pointers within the structure are actually offsets into memory where the data actually lies. The offsets are from the start of the BLOB data. This requires you to fix up the pointers to reference the absolute memory location before you can actually access the data. Figure 10-2 shows the *HOSTENT* structure and memory layout returned. The DNS query is performed on the host name "riven," which has a single IP address and no aliases. Each field in the structure has the offset value. In order to correct this so the fields reference the right location, you need to add the offset value to the address of the head of the *HOSTENT* structure. This needs to be performed on the *h_name* , *h_aliases* , and *h_addr_list* fields. Additionally, the *h_aliases* and *h_addr_list* fields are an array of pointers. Once you obtain the correct pointer to the array of pointers, each 32-bit field in the references location is made up of offsets. If you take a look at the *h_addr_list* field in Figure 10-2, you'll see that the initial offset is 16 bytes, which references the byte after the end of the *HOSTENT* structure. This is the array of pointers to the 4-byte IP address. However, the first pointer in the array is an offset of 28 bytes. To reference the correct location, take the address of the *HOSTENT* structure and add 28 bytes, which points to a 4-byte location with the data 0x9D36B9BA, which is the IP address 157.54.185.186. You then take the 4 bytes after the entry with the offset of 28 bytes, which is 0. This signifies the end of the array of pointers. If multiple IP addresses were associated with this host name, another offset would be present and you would fix the pointer exactly as in the first case. The same procedure is done to fix the *h_aliases* pointer and the array of pointers it references. In this example, there are no aliases for our host. The first entry in the array is 0, which indicates that you don't have to do any further work for that field. The last field is the *h_name* field, which is easy to correct; simply add the offset to the address of the *HOSTENT* structure, and it points to the start of a null-terminated string.

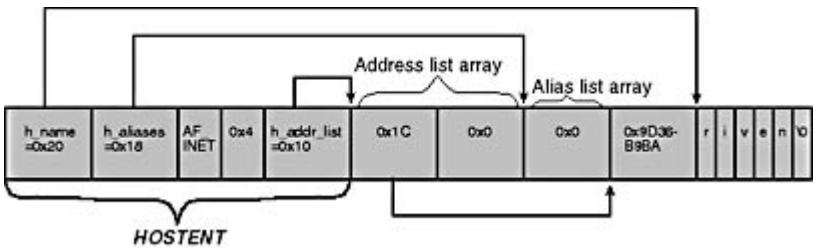

Figure 10-2. HOSTENT *BLOB format*

The code needed to fix these offsets into real addresses is simple, although quite a bit of pointer arithmetic is involved. To fix the *h_name* field, a simple offset adjustment such as the following will do:

```
hostent->h_name = (PCHAR)((DWORD_PTR)hostent->h_name) + (PCHAR)hostent;
```

To fix the array of pointers, as in the *h_aliases* and *h_addr_list* fields, requires a bit more code, but only to traverse the array and fix the references until a null entry is hit. The code looks like this:

```
PCHAR  *addr;

if (hostent->h_aliases)
{
    addr = hostent->h_aliases = (PCHAR)((DWORD_PTR)hostent->h_aliases +
(PCHAR)hostent);
   while (addr)
```

```
   {
        addr = (PCHAR)((DWORD_PTR)addr + (PCHAR *)hostent);
        addr++;
    }
}
```

The code simply steps through each array entry and adds the starting address of the *HOSTENT* structure to the given offset, which becomes the value for that entry. Of course, once you hit an array entry whose value is 0, you stop. The same process needs to be applied to the *h_addr_list* field as well. Once the offsets are fixed, you can use the *HOSTENT* structure as you normally would.

Conclusion

The RNR functions might seem overly complicated, but they offer great flexibility in writing client/server applications. The real limitation of name registration lies with the name space. It's rather amazing that even with the popularity of TCP/IP, the only name resolution method available has been DNS, which is not very flexible. With Windows 2000 and the Windows NT domain space, a persistent, protocol-independent method of name resolution is available, offering the necessary flexibility to write robust applications. Additionally, other name spaces (such as SAP) are available for IPX/SPX-based applications, offering many of the same capabilities of NTDS (except for protocol independence).

Chapter 11

Multicasting

Multicasting is a relatively new technology that allows data to be sent from one member and then replicated to many others without creating a network traffic nightmare. This technology was developed as an alternative to broadcasting, which can negatively impact network bandwidth if used extensively. Multicast data is replicated to a network only if processes running on workstations in that network are interested in that data. Not all protocols support the notion of multicasting—on Win32 platforms, only two protocols accessible from Winsock are capable of supporting multicast traffic: IP and ATM. This chapter presents the information necessary to understand multicasting in general, as well as how multicasting specifically applies to these two protocols.

First we will cover the basic semantics of multipoint networking, which includes the various possible types of multicasting as well as the basic characteristics of IP and ATM multicasting. Following this, the specifics of IP and ATM will be presented in their own sections. Last we will cover the API calls for multicasting in Winsock 1 and Winsock 2. Winsock has supported IP multicasting for quite some time, starting with Winsock 1. In Winsock 2, the multicasting interface was extended to become protocol-independent.

The platforms that support multicasting are Windows CE 2.1, Windows 95, Windows 98, Windows NT 4, and Windows 2000. IP multicast support for Windows CE is new to version 2.1 and unavailable in earlier versions. All of the platforms supporting multicasting are capable of IP multicasting. However, since native ATM Winsock support is available only in Windows 98 and Windows 2000, only these two platforms support ATM multicasting natively. Note that this doesn't preclude you from developing IP multicasting applications that run on an ATM network. It only means that you cannot write *native* ATM multicasting code. We will discuss this in more detail in the IP multicasting section. Another multicasting support issue is that some older network cards are not capable of sending and receiving on the IP multicast addresses. Most NICs manufactured in the last couple of years do support IP multicasting, but this certainly isn't always true.

Multicast Semantics

Multicasting has two important properties: the control plane and the data plane. The *control plane* defines the way in which group membership is organized. The *data plane* refers to the way in which data is propagated among the members. Either one of these properties can be rooted or nonrooted. In a *rooted* control plane, there is a special member of the multicast group known as the *c_root*. Each of the remaining group members is known as a *c_leaf*. In most cases, the c_root establishes the multipoint group by initiating connections to any number of c_leafs. In some cases, a c_leaf might request membership to a given multipoint group at a later time. Note that there can be only one root node for a given group. The ATM protocol is an example of a rooted control plane.

A *nonrooted* control plane allows anyone to join a group without exception. In this situation, all group members are c_leaf nodes. Each member has the power to join a multipoint group. You can impose your own group membership scheme in a nonrooted control plane (this will in effect make one node a c_root) by implementing your own group membership protocol. However, your group membership scheme is still built upon a nonrooted control plane. IP multicasting is an example of a nonrooted control plane. Figure 11-1 illustrates the difference between rooted and nonrooted control planes. In the rooted control plane on the left, the c_root must explicitly ask each c_leaf to join the group, while in the nonrooted scheme on the right anyone can join the group.

The data plane also can be rooted or nonrooted. A rooted data plane has a participant called the *d_root*. The transfer of data occurs only between d_root and all other members of the multipoint session, who are each referred to as a *d_leaf*. The traffic can be either unidirectional or bidirectional, but a rooted data plane implies that data sent from one d_leaf will be received only by the d_root, while data sent from the d_root will be received by each d_leaf. ATM is an example of a rooted data plane. Figure 11-2 illustrates the difference between rooted and nonrooted data planes. In the rooted data plane on the left, data *abc* from the d_root is propagated to every d_leaf. Data *xyz* sent from a d_leaf is received only by the d_root. This contrasts with the nonrooted example on the right, in which data *abc* and *xyz* are propagated to every member, no matter who sent the data.

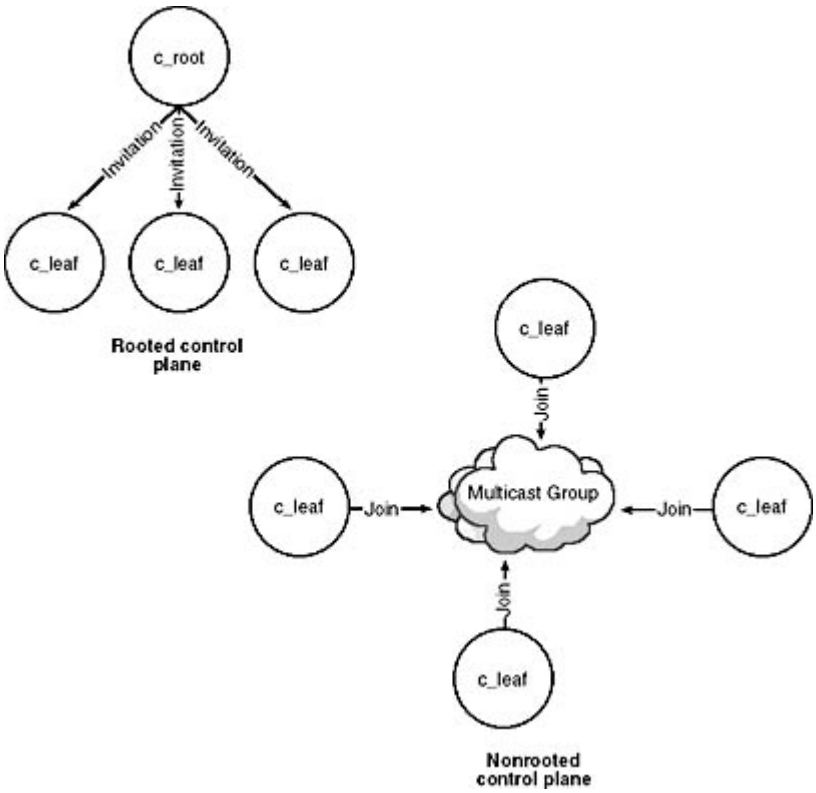

Figure 11-1. *Rooted and nonrooted control planes*

Finally, in a nonrooted data plane all group members can send data to all other members of the group. A block of data sent from a group member is delivered to all other members, and all recipients can send data back. There are no restrictions on who can receive or send data. Again, IP multicasting is nonrooted in the data plane.

So we see that ATM multicasting is rooted in the control and data planes, while IP multicasting is nonrooted in

both planes. Combinations other than these two can exist. For example, it is possible to have a rooted control plane in which one node decides who can join the group and also have a nonrooted data plane in which data sent from any member is seen by all other members. However, none of the currently supported protocols under Winsock behave in this manner.

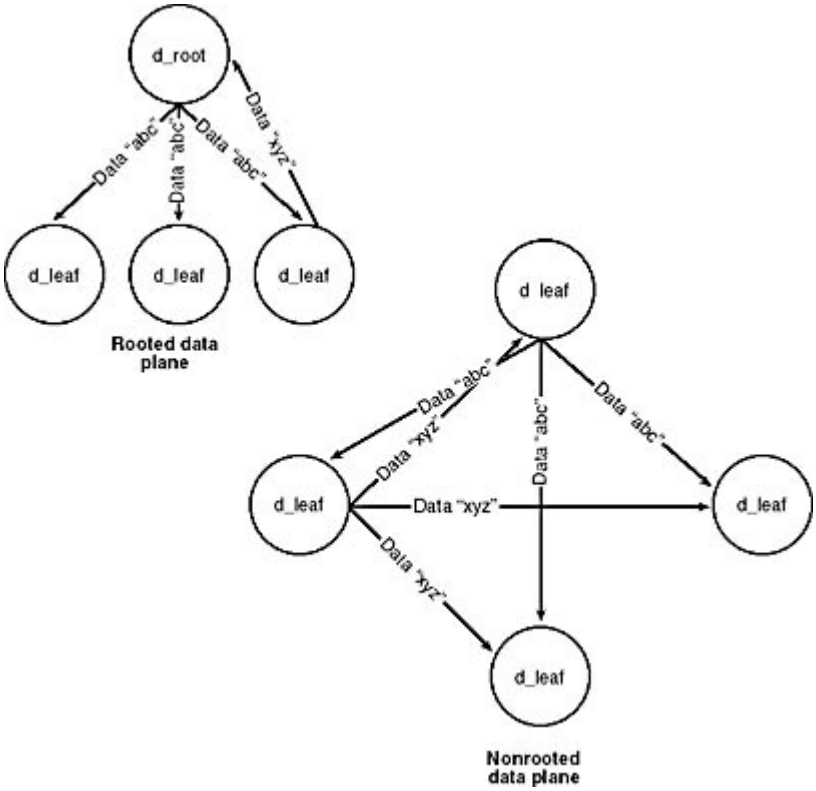

Figure 11-2. *Rooted and nonrooted data planes*

Finding Multicast Properties

In Chapter 5, we discussed how to enumerate protocol entries and determine their properties. All the pertinent multipoint information about a protocol is also available from the protocol's entry in the catalog. The *dwServiceFlags1* entry in the *WSAPROTOCOL_INFO* structure returned by *WSAEnumProtocols* contains several bits we're interested in. If the *XP1_SUPPORT_MULTIPOINT* bit is set, the protocol entry supports multicasting. Then, if the *XP1_MULTIPOINT_CONTROL_PLANE* bit is set, the protocol supports a rooted control plane; otherwise, it is nonrooted. If the *XP1_MULTIPOINT_DATA_PLANE* bit is set, the protocol supports a rooted data plane. Likewise, if the bit is 0, the protocol supports only a nonrooted data plane.

IP Multicasting

IP multicasting relies on a special group of addresses known as multicast addresses. It is this group address that names a given group. For example, if five nodes all want to communicate with one another via IP multicast, they all join the same group address. Once they are joined, any data sent by one node is replicated to every member of the group, including the node that sent the data. A multicast IP address is a class D IP address in the range 224.0.0.0 through 239.255.255.255. A number of these addresses are reserved for special purposes. For example, 224.0.0.0 is never used, 224.0.0.1 represents all systems on the subnet, and 224.0.0.2 represents all routers on the subnet. These last two special addresses are reserved for use by the IGMP protocol, which we will present shortly. For a comprehensive list of reserved addresses, take a look at RFC 1700, which is a list of network resources reserved for a particular use. The Internet Assigned Numbers Authority (IANA) maintains this list. Table 11-1 lists a few of the addresses currently marked as reserved. In actuality, you can use any address except for the first three reserved multicast addresses, as they are used by routers on the network. Refer to RFC 1700 for the exact multicast address assignments.

Table 11-1. *Multicast addresses*

Multicast Address	*Use*
224.0.0.0	Base address (reserved)
224.0.0.1	All systems on this subnet
224.0.0.2	All routers on this subnet
224.0.1.1	Network time protocol
224.0.0.9	RIP version 2 group address
224.0.1.24	WINS server group address

Because multicasting had not been envisioned when TCP/IP was developed, a number of accommodations had to be made to allow IP to support it. For example, we have already discovered that IP requires a set of special addresses to be set aside for multicast traffic. In addition, a special protocol was introduced to manage multicast clients and their membership in a group. Imagine if two workstations on separate subnets want to join a single multicast group. How is this implemented over IP? You can't simply broadcast the data to the multicast address everywhere because the network would become flooded with broadcast data in no time. The Internet Gateway Management Protocol (IGMP) was developed to signal routers that a machine on the network is interested in data destined for a given group.

Internet Gateway Management Protocol

Multicasting hosts use IGMP to notify routers that a computer on the router's subnet wants to join a particular multicast group. IGMP is the backbone of IP multicasting. For it to work correctly, all routers between two multicasting nodes must support IGMP. For example, if machines A and B join the multicast group 224.1.2.3, and there are three routers between the two, all three routers must be IGMP-enabled in order for successful communication to occur. Any non-IGMP-enabled router simply drops received multicast data. When an application joins a multicast group, an IGMP "join" command is sent to the all-routers address (224.0.0.2) on the subnet. This command notifies the router that it has clients interested in a particular multicast address. Thus, if the router receives data destined for that multicast address, it forwards it to the subnet with the multicast client.

Additionally, when an endpoint joins a multicast group it specifies a time-to-live (TTL) parameter that indicates how many routers the endpoint's multicast application is willing to traverse to send and receive data. For example, if you write an IP multicast application that joins group X with a TTL of 2, a join command is sent to the all-routers group on the local subnet. The routers on that subnet pick up the command, indicating that it should forward multicast data destined for that address. The router decrements the TTL by 1 and passes the join command on to its neighboring networks. The routers on those networks do the same upon receipt of the command. At this point, those routers decrement the TTL again, which now makes the TTL value 0, and the command is no longer propagated. Because of this, TTL limits how far multicast data will be replicated.

Once a router has one or more multicast groups registered by workstations, it periodically sends a "group query" message to the all-hosts group (224.0.0.1) for each multicast address that it was notified of by a join command. If clients on that network are still using that multicast address, they respond with another IGMP message so that the

router knows to keep forwarding data related to that address; otherwise, the router stops forwarding any data for that address. Even if the client explicitly leaves that multicast group using either of the Winsock methods for leaving a group, the filter set on the router is not immediately removed.

Unfortunately this can be disastrous: a client can drop membership to multicast group A and immediately join group B. Until the router performs a group query and receives no responses, the router will be forwarding data destined for both multicast groups A and B to the network. This is especially bad if the sum of the transmitted data associated with both groups is more than the network bandwidth will allow. This is where IGMP version 2 comes into play. The newest version allows for a client to explicitly send a "leave" message to the router to immediately stop forwarding data for the given multicast address. Of course, the router maintains a reference count of the number of clients for each particular address so that until all clients on the subnet release a particular address, data destined for that address continues to be propagated.

Windows 98 and Windows 2000 natively support IGMP version 2. For Windows 95, the latest Winsock 2 update also includes IGMP version 2. In Windows NT 4, Service Pack 4 includes support for IGMP version 2. Previous Service Packs and the base OS itself supported only version 1. If you want to read the complete specifications on IGMP version 1 or version 2, consult RFC 1112 or RFC 2236, respectively.

IP Leaf Nodes

The process of joining an IP multicast group is simple, as every node is a leaf and therefore performs the same steps to join a group. Because IP multicasting was available in Winsock 1 as well as Winsock 2, two methods of API calls can be performed to accomplish the same thing. Here are the basic steps for IP multicasting in Winsock 1:

1. Use the *socket* function to create a socket with the *AF_INET* address family and the *SOCK_DGRAM* socket type. No special flags are required to indicate a multicast socket because the *socket* function has no flags parameter.

2. Bind the socket to a local port if you want to receive data from the group.

3. Call the *setsockopt* function with the *IP_ADD_MEMBERSHIP* option with the address structure of the group you want to join.

If you are using Winsock 2, steps 1 and 2 are the same, but step 3 is to call the *WSAJoinLeaf* function to add yourself to the group. These two functions and their differences are discussed in greater detail in the section on multicasting with Winsock later in this chapter.

One reminder: an application does not need to join an IP multicast group if it is only sending data. When sending data to a multicast group, the packet on the network is the same as a normal UDP packet except that the destination address is a special multicast address. If you need to receive multicast data, you are required to join the group. Other than the group membership requirements, however, IP multicasting behaves just like normal UDP protocol: it is connectionless, nonreliable, and so on.

IP Multicast Implementation

IP multicasting is supported by all the Windows platforms (with the exception of Windows CE prior to version 2.1). As a result, IP multicasting is implemented somewhat differently on each platform. We mentioned earlier that the network card in use must support multicasting. It does this by providing the ability to add multicast filters in hardware to the interface. Multicast IP addresses use a special MAC address that contains the encoded IP address so that network cards can easily determine whether an incoming packet is multicast and can also determine which multicast IP address the packet is destined for just by examining the MAC header. The exact encoding mechanism is discussed in RFC 1700. We will not cover it in this book, as it doesn't pertain to the Winsock API.

Windows 95 and Windows NT 4, however, implement multicasting by putting the network card in promiscuous mode. This means that the card is picking up all packets arriving on the wire and the network driver is examining the MAC headers, looking for multicast data destined for a group that a process on the workstation is a member of. Because this filtering is being performed by software, it is not as optimal as having the hardware filter packets for you. Windows 98 and Windows 2000 are implemented differently. Both take advantage of the card's ability to add filters, which is optimal because this is performed by the hardware. Most network cards can support 16 or 32 hardware filters. The only limitation on Windows 98 is that once all the hardware filters are set, no more multicast groups can be joined by a process on the machine. If the hardware limit is exceeded, the multicast join operation fails with *WSAENOBUFS*. Windows 2000 is more robust. Once all the hardware filters are exhausted, it puts the

network card in promiscuous mode and operates in a manner similar to Windows 95 and Windows NT 4.

ATM Multicasting

Native ATM through Winsock also supports multicasting, which offers significantly different capabilities than IP multicasting. Remember that ATM supports rooted control and data planes, meaning that when a multicast server, or c_root, is established, it has control over who is allowed to join the group as well as how data is transmitted within the group.

One important distinction is that on an ATM network, IP over ATM can be enabled. This configuration allows the ATM network to emulate an IP network by mapping IP addresses to ATM native addresses. With IP over ATM enabled, you have a choice of using IP multicasting, which will be translated appropriately to the ATM layer, or using the native ATM multicasting capabilities, which we present in this section. The behavior of IP multicasting on ATM when configured for IP over ATM should be the same, as it appears that you are on an IP network. The only exception is that IGMP is not present because all multicast calls are translated into ATM native commands. Additionally, it is possible to configure an ATM network with one or more LAN emulation (LANE) networks. The purpose of a LANE is to make the ATM appear as a "regular" network capable of multiple protocols such as IPX/SPX, NetBEUI, TCP/IP, IGMP, ICMP, and so on. In this situation, IP multicasting does in fact look every bit like IP multicasting on an Ethernet network, which means IGMP is present as well.

We mentioned that ATM supports rooted control planes and rooted data planes. This means that when you create a multicast group, you establish a root node that "invites" leaf nodes to join the multicast group. In Windows 2000, only root-initiated joins are currently supported, meaning that a leaf cannot request to be added to a group. Additionally, the root node (as a rooted data plane) sends data in one direction only, from the root to the leaves.

One startling contrast between ATM and IP multicasting is that ATM needs no special addresses. All that is required is that the root have knowledge of the addresses of each leaf that it will invite. Also, only one root node can be in a multicast group. If another ATM endpoint starts inviting the same leaves, this association becomes a separate group from the first.

ATM Leaf Nodes

Creating a leaf node in a multicast group is straightforward. On an ATM network, the leaf must listen for an invitation from a root to join a group. The necessary steps are outlined below.

1. Using the *WSASocket* function, create a socket of address family *AF_ATM* with the flags *WSA_FLAG_MULTIPOINT_C_LEAF* and *WSA_FLAG_MULTIPOINT_D_LEAF*
2. Bind the socket to the local ATM address and port with the *bind* function.
3. Call *listen*.
4. Wait for an invitation using *accept* or *WSAAccept*. Depending on the I/O model you use, this will differ. (See Chapter 8 for a more thorough description of the Winsock I/O models.)

Once the connection is established, the leaf node can receive data sent from the root. Remember that with ATM multicasting, the data flow is one way: from the root to the leaves.

> NOTE
>
> Windows 98 and Windows 2000 currently support only a single ATM leaf node on the system at any given time. That is, only a single process on the entire system can be a leaf member of any ATM point-to-multipoint session.

ATM Root Nodes

Creating a root node is even easier than creating an ATM leaf. The basic steps are

1. Using the *WSASocket* function, create a socket of address family *AF_ATM* with the flags *WSA_FLAG_MULTIPOINT_C_ROOT* and *WSA_FLAG_MULTIPOINT_D_ROOT*

2. Call *WSAJoinLeaf* with the ATM address for each endpoint you want to invite.

   A root node can invite as many endpoints as it wants, but it must issue a separate *WSAJoinLeaf* call for each.

Multicasting with Winsock

Now that you are familiar with the two types of multicasting available on Windows platforms, let's take a look at the API calls provided by Winsock to make multicasting possible. Winsock offers two different methods for IP multicasting that are based on the version of Winsock you use. The first method, available in Winsock 1, is to use socket options to join a group. Winsock 2 introduces a new function for joining a multicast group, *WSAJoinLeaf*, which is independent of the underlying protocol. The Winsock 1 method, which we will cover first, is the most widely used method, especially because it was derived from Berkeley sockets.

Winsock 1 Multicasting

Winsock 1 implements joining and leaving an IP multicast group by using *setsockopt* with the *IP_ADD_MEMBERSHIP* and *IP_DROP_MEMBERSHIP* options. When using either of the socket options, you must pass in an *ip_mreq* structure, which is defined as follows:

```
struct ip_mreq
{
    struct in_addr  imr_multiaddr;
    struct in_addr  imr_interface;
};
```

The *imr_multiaddr* field specifies the multicast group to join, while *imr_interface* specifies the local interface to send multicast data out on. If you specify *INADDR_ANY* for *imr_interface*, the default interface is used; otherwise, specify the IP address of the local interface to be used if there is more than one. The following code example illustrates joining the multicast group 234.5.6.7.

```
SOCKET            s;
struct ip_mreq    ipmr;
SOCKADDR_IN       local;
int               len = sizeof(ipmr);

s = socket(AF_INET, SOCK_DGRAM, 0);

local.sin_family = AF_INET;
local.sin_addr.s_addr = htonl(INADDR_ANY);
local.sin_port = htons(10000);

ipmr.imr_multiaddr.s_addr = inet_addr("234.5.6.7");
ipmr.imr_interface.s_addr = htonl(INADDR_ANY);

bind(s, (SOCKADDR *)&local, sizeof(local));
setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP,
    (char *)&ipmr, &len);
```

To leave the multicast group, simply call *setsockopt* with *IP_DROP_MEMBERSHIP*, passing an *ip_mreq* structure with the same values that you used to join the group, as in the following code:

```
setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP,
    (char *)&ipmr, &len);
```

Winsock 1 IP multicast example

In Figure 11-3 we provide a simple IP multicast sample program that joins a specified group and then acts as a sender or receiver, depending on the command line arguments. This example is written so that a sender only sends, whereas a receiver simply waits in a loop for incoming data. This design doesn't illustrate the fact that if the sender sends data to the multicast group and is also listening for data as a member of that group, the data sent will be looped back to the sender's own incoming data queue. This is known as the multicast loopback, and it occurs only when you use IP multicasting. We will discuss how to disable this loopback later in this chapter.

Figure 11-3. *The Winsock 1.1 multicast example*

```
// Module name: Mcastws1.c

#include <windows.h>
#include <winsock.h>

#include <stdio.h>
#include <stdlib.h>

#define MCASTADDR     "234.5.6.7"
#define MCASTPORT      25000
#define BUFSIZE        1024
#define DEFAULT_COUNT  500

BOOL  bSender = FALSE,      // Act as sender?
      bLoopBack = FALSE;    // Disable loopback?

DWORD dwInterface,          // Local interface to bind to
      dwMulticastGroup,     // Multicast group to join
      dwCount;              // Number of messages to send/receive

short iPort;                // Port number to use

//
// Function: usage
//
// Description:
//    Print usage information and exit
//
void usage(char *progname)
{
    printf("usage: %s -s -m:str -p:int -i:str -l -n:int\n",
        progname);
    printf(" -s     Act as server (send data); otherwise\n");
    printf("          receive data.\n");
    printf(" -m:str Dotted decimal multicast IP addres to join\n");
    printf("          The default group is: %s\n", MCASTADDR);
    printf(" -p:int Port number to use\n");
    printf("          The default port is: %d\n", MCASTPORT);
    printf(" -i:str Local interface to bind to; by default \n");
    printf("          use INADDRY_ANY\n");
    printf(" -l     Disable loopback\n");
    printf(" -n:int Number of messages to send/receive\n");
    ExitProcess(-1);
}
```

```
//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some global flags,
//    depending on the values
//
void ValidateArgs(int argc, char **argv)
{
    int      i;

    dwInterface = INADDR_ANY;
    dwMulticastGroup = inet_addr(MCASTADDR);
    iPort = MCASTPORT;
    dwCount = DEFAULT_COUNT;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 's':   // Sender
                    bSender = TRUE;
                    break;
                case 'm':   // Multicast group to join
                    if (strlen(argv[i]) > 3)
                        dwMulticastGroup = inet_addr(&argv[i][3]);
                    break;
                case 'i':   // Local interface to use
                    if (strlen(argv[i]) > 3)
                        dwInterface = inet_addr(&argv[i][3]);
                    break;
                case 'p':   // Port number to use
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 'l':   // Disable loopback?
                    bLoopBack = TRUE;
                    break;
                case 'n':   // Number of messages to send/recv
                    dwCount = atoi(&argv[i][3]);
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
    }
    return;
}

//
```

```
// Function: main
//
// Description:
//    Parse the command line arguments, load the Winsock library,
//    create a socket, and join the multicast group. If this program
//    is started as a sender, begin sending messages to the
//    multicast group; otherwise, call recvfrom() to read messages
//    sent to the group.
//
int main(int argc, char **argv)
{
    WSADATA             wsd;
    struct sockaddr_in  local,
                        remote,
                        from;
    struct ip_mreq      mcast;
    SOCKET              sockM;
    TCHAR               recvbuf[BUFSIZE],

                        sendbuf[BUFSIZE];
    int                 len = sizeof(struct sockaddr_in),
                        optval,
                        ret;
    DWORD               i=0;

    // Parse the command line, and load Winsock
    //
    ValidateArgs(argc, argv);

    if (WSAStartup(MAKEWORD(1, 1), &wsd) != 0)
    {
        printf("WSAStartup failed\n");
        return -1;
    }
    // Create the socket. In Winsock 1, you don't need any special
    // flags to indicate multicasting.
    //
    if ((sockM = socket(AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET)
    {
        printf("socket failed with: %d\n", WSAGetLastError());
        WSACleanup();
        return -1;
    }
    // Bind the socket to the local interface. This is done so
    // that we can receive data.
    //
    local.sin_family = AF_INET;
    local.sin_port   = htons(iPort);
    local.sin_addr.s_addr = dwInterface;

    if (bind(sockM, (struct sockaddr *)&local,
        sizeof(local)) == SOCKET_ERROR)
    {
```

```
        printf("bind failed with: %d\n", WSAGetLastError());
        closesocket(sockM);
        WSACleanup();
        return -1;
    }

    // Set up the im_req structure to indicate what group we want
    // to join as well as the interface
    //
    remote.sin_family      = AF_INET;
    remote.sin_port        = htons(iPort);
    remote.sin_addr.s_addr = dwMulticastGroup;

    mcast.imr_multiaddr.s_addr = dwMulticastGroup;
    mcast.imr_interface.s_addr = dwInterface;

    if (setsockopt(sockM, IPPROTO_IP, IP_ADD_MEMBERSHIP,
        (char *)&mcast, sizeof(mcast)) == SOCKET_ERROR)
    {
        printf("setsockopt(IP_ADD_MEMBERSHIP) failed: %d\n",
            WSAGetLastError());
        closesocket(sockM);
        WSACleanup();
        return -1;
    }
    // Set the TTL to something else. The default TTL is 1.
    //
    optval = 8;
    if (setsockopt(sockM, IPPROTO_IP, IP_MULTICAST_TTL,
        (char *)&optval, sizeof(int)) == SOCKET_ERROR)
    {
        printf("setsockopt(IP_MULTICAST_TTL) failed: %d\n",
            WSAGetLastError());
        closesocket(sockM);
        WSACleanup();
        return -1;
    }
    // Disable the loopback if selected. Note that in Windows NT 4
    // and Windows 95 you cannot disable it.
    //
    if (bLoopBack)
    {
        optval = 0;
        if (setsockopt(sockM, IPPROTO_IP, IP_MULTICAST_LOOP,
            (char *)&optval, sizeof(optval)) == SOCKET_ERROR)
        {
            printf("setsockopt(IP_MULTICAST_LOOP) failed: %d\n",
                WSAGetLastError());
            closesocket(sockM);
            WSACleanup();
            return -1;
        }
    }
```

```
    if (!bSender)           // Client
    {
        // Receive some data
        //
        for(i = 0; i < dwCount; i++)
        {

            if ((ret = recvfrom(sockM, recvbuf, BUFSIZE, 0,
                (struct sockaddr *)&from, &len)) == SOCKET_ERROR)
            {
                printf("recvfrom failed with: %d\n",
                    WSAGetLastError());
                closesocket(sockM);
                WSACleanup();
                return -1;
            }
            recvbuf[ret] = 0;
            printf("RECV: '%s' from <%s>\n", recvbuf,
                inet_ntoa(from.sin_addr));
        }
    }
    else                    // Server
    {
        // Send some data
        //
        for(i = 0; i < dwCount; i++)
        {
            sprintf(sendbuf, "server 1: This is a test: %d", i);
            if (sendto(sockM, (char *)sendbuf, strlen(sendbuf), 0,
                (struct sockaddr *)&remote,
                sizeof(remote)) == SOCKET_ERROR)
            {
                printf("sendto failed with: %d\n",
                    WSAGetLastError());
                closesocket(sockM);
                WSACleanup();
                return -1;
            }
            Sleep(500);
        }
    }
    // Drop group membership
    //
    if (setsockopt(sockM, IPPROTO_IP, IP_DROP_MEMBERSHIP,
        (char *)&mcast, sizeof(mcast)) == SOCKET_ERROR)
    {
        printf("setsockopt(IP_DROP_MEMBERSHIP) failed: %d\n",
            WSAGetLastError());
    }
    closesocket(sockM);

    WSACleanup();
```

```
    return 0;
}
```

One note of caution when you implement the Winsock 1 method of multicasting: use the correct header file and library for linking. If you load the Winsock 1.1 library, you should include *Winsock.h* and link with *Wsock32.lib.* If you're loading version 2 or later, include *Winsock2.h* and *Ws2tcpip.h* and link with *Ws2_32.lib*. This is necessary because two different sets of values exist for the constants *IP_ADD_MEMBERSHIP*, *IP_DROP_MEMBERSHIP*, *IP_MULTICAST_IF*, and *IP_MULTICAST_LOOP*. The original specification for these values—by Stephen Deering—was never officially integrated into the Winsock specification. As a result, these values changed in the Winsock 2 specification. Of course, if you are using the older Winsock version, linking with *wsock32.lib* takes care of everything, so the constants translate into the correct values when run on a Winsock 2 machine.

Winsock 2 Multicasting

Winsock 2 multicasting is a bit more complicated than Winsock 1 multicasting, but it offers facilities to support protocols that offer additional features, such as Quality of Service (QOS). Additionally, it provides support for protocols supporting rooted multicast schemes. Socket options are no longer used to initiate group membership; instead, a new function, *WSAJoinLeaf*, is introduced in its place. The prototype is as follows:

```
SOCKET WSAJoinLeaf(
    SOCKET s,
    const struct sockaddr FAR * name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
    DWORD dwFlags
);
```

The first parameter, *s*, is the socket handle returned from *WSASocket*. The socket passed in must be created with the appropriate multicast flags; otherwise, *WSAJoinLeaf* will fail with the error *WSAEINVAL*. Remember that you need to specify two multipoint flags: one to indicate whether this socket will be rooted or nonrooted in the control plane and another to indicate whether this socket will be rooted or nonrooted in the data plane. The flags for specifying the control plane are *WSA_FLAG_MULTIPOINT_C_ROOT* and *WSA_FLAG_MULTIPOINT_C_LEAF*, while the flags for the data plane are *WSA_FLAG_MULTIPOINT_D_ROOT* and *WSA_FLAG_MULTIPOINT_D_LEAF*. The second parameter is the *SOCKADDR* structure specific to the protocol in use. For rooted control schemes (ATM, for example), this address specifies the client to invite, while for nonrooted control schemes (IP, for example), this address is that of the multicast group that a host wants to join. The *namelen* parameter is exactly that: the length in bytes of the *name* parameter. The *lpCallerData* parameter is used to transfer a buffer of data to the peer upon session establishment, while *lpCalleeData* indicates a buffer to be transmitted back from the peer upon session establishment. Note that these two parameters are not implemented on the current Windows platforms and should be set to *NULL*. The *lpSQOS* parameter specifies a *FLOWSPEC* structure indicating the required bandwidth for the application. (Chapter 12 covers QOS in greater detail.) The *lpGQOS* parameter is ignored, as none of the current Windows platforms support socket groups. The last parameter, *dwFlags*, specifies whether this host will be sending data, receiving data, or both. The possible values for this parameter are *JL_SENDER_ONLY*, *JL_RECEIVER_ONLY*, or *JL_BOTH*.

The function returns a *SOCKET* descriptor for the socket bound to the multicast group. If the *WSAJoinLeaf* call is made with an asynchronous (or nonblocking) socket, the returned socket descriptor is *not* usable until after the join operation completes. For example, if the socket is in asynchronous mode from a *WSAAsyncSelect* or *WSAEventSelect* call, the descriptor isn't valid until the original socket *s* receives a corresponding *FD_CONNECT* indication. Note that an *FD_CONNECT* notification is generated only in rooted control schemes in which the *name* parameter indicates a specific endpoint's address. Table 11-2 lists under what circumstances an application will receive an *FD_CONNECT* notification. You can cancel the outstanding join request for these nonblocking modes by calling *closesocket* on the original socket. A root node in a multipoint session might call *WSAJoinLeaf* one or more times in order to add a number of leaf nodes; however, at most one multipoint connection request at a time can be outstanding.

As we mentioned earlier, the join request cannot complete immediately with nonblocking sockets, and the socket descriptor returned is not usable until the request is complete. If the socket is in nonblocking mode through the

ioctlsocket command *FIONBIO*, the call to *WSAJoinLeaf* will not return the *WSAEWOULDBLOCK* error because the function has effectively returned a "successful start" indication. Note that, in an asynchronous I/O model, the only method for receiving notification of a successful start is via the *FD_CONNECT* message. See Chapter 8 for more information on using the *WSAAsyncSelect* and *WSAEventSelect* asynchronous I/O models. Blocking sockets cannot notify an application of either a success or a failure of *WSAJoinLeaf*. In other words, straight nonblocking sockets should probably be avoided, as there is no definitive way of determining whether the multicast join succeeds until the socket is actually used in subsequent Winsock calls (which would fail if the join failed).

The socket descriptor returned by *WSAJoinLeaf* differs depending on whether the input socket is a root node or a leaf node. With a root node, the *name* parameter indicates the address of a specific leaf that is being invited into the multipoint session. For the c_root to maintain leaf membership, *WSAJoinLeaf* returns a new socket handle for the leaf. This new socket has all the same attributes as the root's socket used in the invitation. This includes any asynchronous events registered through the asynchronous I/O models, such as *WSAEventSelect* and *WSAAsyncSelect*. However, these new sockets should be used only to receive *FD_CLOSE* notifications from the leaf. Any data to be sent to the multipoint groups should be sent using the c_root socket. In some cases, you can send data on the socket returned from *WSAJoinLeaf*, but only the leaf corresponding to that socket will receive the data. The ATM protocol allows this. Finally, to remove a leaf node from the multipoint session, the root simply has to call *closesocket* on the socket corresponding to that leaf.

On the other hand, when *WSAJoinLeaf* is invoked with a leaf node, the *name* parameter specifies the address of either a root node or a multipoint group. The former case specifies a leaf-initiated join, which isn't currently supported by any protocol. (However, the ATM UNI 4.0 specification will support this.) The latter case is how IP multicasting operates. In either case, the socket handle returned from *WSAJoinLeaf* is simply the same socket handle passed as *s*. In the case in which the call to *WSAJoinLeaf* is a leaf-initiated join, the root node listens for incoming connections using the *bind*, *listen*, and *accept*/*WSAAccept* methods that you normally expect for a server. When the application wants to remove itself from the multipoint session, *closesocket* is called on the socket that terminates membership (and also frees socket resources). Table 11-2 summarizes the actions that are carried out depending on the type of control plane and the parameters that are passed for the socket and name parameters. These actions include whether a new socket descriptor is returned upon successful function call as well as whether the application will receive an *FD_CONNECT* notification.

Table 11-2. WSAJoinLeaf *actions*

Control Plane	s	Name	*Action*	*Receives* FD_CONNECT *Notification?*	*Returned Socket Descriptor*
Rooted	c_root	Address of leaf	Root is inviting a leaf.	Yes	Used for *FD_CLOSE*notification and for sending private data to that leaf only
	c_leaf	Address of root	Leaf is initiating a connection to root.	Yes	Is a duplicate of *s*
Nonrooted	c_root	N/A	Not a possible combination.	N/A	N/A
	c_leaf	Address of group	Leaf joins a group.	No	Is a duplicate of *s*

When an application calls the *accept* or the *WSAAccept* function either to wait for a root-initiated invitation or, as root, to wait for join requests from leaves, the function returns a socket that is a c_leaf socket descriptor just like those that *WSAJoinLeaf* returns. To accommodate protocols that allow both root-initiated and leaf-initiated joins, it is acceptable for a c_root socket that is already in listening mode to be used as an input to *WSAJoinLeaf*.

Once the call to *WSAJoinLeaf* is made, a new socket handle is returned. This handle is not used to send and receive data. The original socket handle returned from *WSASocket* that is passed into *WSAJoinLeaf* is used on all sending and receiving operations. The new handle indicates that you are a member of the given multicast group as long as the handle is valid. Calling *closesocket* on the new handle will terminate your application's membership to the multicast group. Calling *closesocket* on a c_root socket causes all associated c_leaf nodes using an asynchronous I/O model to receive an *FD_CLOSE* notification.

Winsock 2 IP multicast example

Figure 11-4 contains Mcastws2.c, which illustrates how to join and leave an IP multicast group using *WSAJoinLeaf*. This example is simply the Winsock 1 IP multicast example, Mcastws1.c, except that the join/leave calls have been

converted to use *WSAJoinLeaf*.

Figure 11-4. *IP multicast example*

```
// Module: Mcastws2.c
//
#include <winsock2.h>
#include <ws2tcpip.h>

#include <stdio.h>
#include <stdlib.h>

#define MCASTADDR     "234.5.6.7"
#define MCASTPORT      25000
#define BUFSIZE        1024
#define DEFAULT_COUNT  500

BOOL  bSender = FALSE,      // Act as a sender?
      bLoopBack = FALSE;    // Disable loopback?

DWORD dwInterface,          // Local interface to bind to
      dwMulticastGroup,     // Multicast group to join
      dwCount;              // Number of messages to send/receive

short iPort;                // Port number to use

//
// Function: usage

//
// Description:
//    Print usage information and exit
//
void usage(char *progname)
{
    printf("usage: %s -s -m:str -p:int -i:str -l -n:int\n",
        progname);
    printf("   -s        Act as server (send data); otherwise\n");
    printf("              receive data.\n");
    printf("   -m:str    Dotted decimal multicast IP addres "
       "to join\n");
    printf("              The default group is: %s\n", MCASTADDR);
    printf("   -p:int    Port number to use\n");
    printf("              The default port is: %d\n", MCASTPORT);
    printf("   -i:str    Local interface to bind to; by default \n");
    printf("              use INADDRY_ANY\n");
    printf("   -l        Disable loopback\n");
    printf("   -n:int    Number of messages to send/receive\n");
    ExitProcess(-1);
}

//
// Function: ValidateArgs
```

```
//
// Description:
//    Parse the command line arguments, and set some global flags,
//    depending on the values
//
void ValidateArgs(int argc, char **argv)
{
    int      i;

    dwInterface = INADDR_ANY;
    dwMulticastGroup = inet_addr(MCASTADDR);
    iPort = MCASTPORT;
    dwCount = DEFAULT_COUNT;

    for(i=1; i < argc ;i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 's':  // Sender
                    bSender = TRUE;
                    break;

                case 'm':  // Multicast group to join
                    if (strlen(argv[i]) > 3)
                        dwMulticastGroup = inet_addr(&argv[i][3]);
                    break;
                case 'i':  // Local interface to use
                    if (strlen(argv[i]) > 3)
                        dwInterface = inet_addr(&argv[i][3]);
                    break;
                case 'p':  // Port to use
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 'l':  // Disable loopback
                    bLoopBack = TRUE;
                    break;
                case 'n':  // Number of messages to send/recv
                    dwCount = atoi(&argv[i][3]);
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
    }
    return;
}

//
// Function: main
```

```
//
// Description:
//    Parse the command line arguments, load the Winsock library,
//    create a socket, and join the multicast group. If this program
//    is run as a sender, begin sending messages to the multicast
//    group; otherwise, call recvfrom() to read messages sent to the
//    group.
//
int main(int argc, char **argv)
{
    WSADATA             wsd;
    struct sockaddr_in  local,
                        remote,
                        from;
    SOCKET              sock, sockM;
    TCHAR               recvbuf[BUFSIZE],
                        sendbuf[BUFSIZE];
    int                 len = sizeof(struct sockaddr_in),
                        optval,
                        ret;
    DWORD               i=0;

    // Parse the command line, and load Winsock
    //
    ValidateArgs(argc, argv);

    if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
    {
        printf("WSAStartup() failed\n");
        return -1;
    }
    // Create the socket. In Winsock 2, you do have to specify the
    // multicast attributes that this socket will be used with.
    //
    if ((sock = WSASocket(AF_INET, SOCK_DGRAM, 0, NULL, 0,
                  WSA_FLAG_MULTIPOINT_C_LEAF
              | WSA_FLAG_MULTIPOINT_D_LEAF
              | WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET)
    {
        printf("socket failed with: %d\n", WSAGetLastError());
        WSACleanup();
        return -1;
    }
    // Bind to the local interface. This is done to receive data.
    local.sin_family = AF_INET;
    local.sin_port   = htons(iPort);
    local.sin_addr.s_addr = dwInterface;

    if (bind(sock, (struct sockaddr *)&local,
        sizeof(local)) == SOCKET_ERROR)
    {
        printf("bind failed with: %d\n", WSAGetLastError());
        closesocket(sock);
```

```
        WSACleanup();
        return -1;
    }
    // Set up the SOCKADDR_IN structure describing the multicast
    // group we want to join
    //
    remote.sin_family      = AF_INET;
    remote.sin_port        = htons(iPort);
    remote.sin_addr.s_addr = dwMulticastGroup;
    //
    // Change the TTL to something more appropriate
    //

    optval = 8;
    if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL,
        (char *)&optval, sizeof(int)) == SOCKET_ERROR)
    {
        printf("setsockopt(IP_MULTICAST_TTL) failed: %d\n",
            WSAGetLastError());
        closesocket(sock);
        WSACleanup();
        return -1;
    }
    // Disable loopback if needed
    //
    if (bLoopBack)
    {
        optval = 0;
        if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP,
            (char *)&optval, sizeof(optval)) == SOCKET_ERROR)
        {
            printf("setsockopt(IP_MULTICAST_LOOP) failed: %d\n",
                WSAGetLastError());
            closesocket(sock);
            WSACleanup();
            return -1;
        }
    }
    // Join the multicast group.  Note that sockM is not used
    // to send or receive data. It is used when you want to
    // leave the multicast group. You simply call closesocket()
    // on it.
    //
    if ((sockM = WSAJoinLeaf(sock, (SOCKADDR *)&remote,
        sizeof(remote), NULL, NULL, NULL, NULL,
        JL_BOTH)) == INVALID_SOCKET)
    {
        printf("WSAJoinLeaf() failed: %d\n", WSAGetLastError());
        closesocket(sock);
        WSACleanup();
        return -1;
    }
```

```
    if (!bSender)           // Receiver
    {
        // Receive data
        //
        for(i = 0; i < dwCount; i++)
        {
            if ((ret = recvfrom(sock, recvbuf, BUFSIZE, 0,
                (struct sockaddr *)&from, &len)) == SOCKET_ERROR)
            {
                printf("recvfrom failed with: %d\n",
                    WSAGetLastError());
                closesocket(sockM);
                closesocket(sock);
                WSACleanup();
                return -1;
            }
            recvbuf[ret] = 0;
            printf("RECV: '%s' from <%s>\n", recvbuf,
                inet_ntoa(from.sin_addr));
        }
    }
    else                    // Sender
    {
        // Send data
        //
        for(i = 0; i < dwCount; i++)
        {
            sprintf(sendbuf, "server 1: This is a test: %d", i);
            if (sendto(sock, (char *)sendbuf, strlen(sendbuf), 0,
                (struct sockaddr *)&remote,
                sizeof(remote)) == SOCKET_ERROR)
            {
                printf("sendto failed with: %d\n",
                    WSAGetLastError());
                closesocket(sockM);
                closesocket(sock);
                WSACleanup();
                return -1;
            }
            Sleep(500);
        }
    }
    // Leave the multicast group by closing sock.
    // For nonrooted control and data plane schemes, WSAJoinLeaf
    // returns the same socket handle that you pass into it.
    //
    closesocket(sock);

    WSACleanup();
    return -1;
}
```

Winsock 2 ATM multicast example

Figure 11-5 contains *Mcastatm.c*, which is a simple ATM multicasting example that illustrates a root-initiated join. The figure does not include the file *Support.c*, which includes some routines that are used by the multicast example but aren't specific to multicasting, such as *GetATMAddress*, which simply returns the ATM address of the local interface.

When you look at the code in Figure 11-5, you will see that the most important section for the multicast root is in the function *Server*, which uses a loop to call *WSAJoinLeaf* once for each client specified on the command line. You'll see that the server keeps an array of sockets for each client that joins; however, in the *WSASend* calls, the main socket is used. If the ATM protocol supported it, the server could choose to listen on the leaf socket (for example, the socket returned from *WSAJoinLeaf*) if it were interested in receiving "side chat" information. In other words, if the client sends data on the connected socket, the server receives it on the corresponding handle returned from *WSAJoinLeaf*. No other clients will receive this data.

On the client side, look at the *Client* function. You'll see that the only requirement is that the client bind to a local interface and await the server's invitation on an *accept* or a *WSAAccept* call. Once the invitation arrives, the newly created socket can be used to receive data sent by the root.

Figure 11-5. *ATM multicast example*

```
// Module: Mcastatm.c

#include "Support.h"

#include <stdio.h>
#include <stdlib.h>


#define BUFSIZE               1024
#define MAX_ATM_LEAF             4

#define ATM_PORT_OFFSET       ((ATM_ADDR_SIZE * 2) - 2)
#define MAX_ATM_STR_LEN       (ATM_ADDR_SIZE * 2)

DWORD  dwAddrCount = 0,
       dwDataCount = 20;
BOOL   bServer = FALSE,
       bLocalAddress = FALSE;
char   szLeafAddresses[MAX_ATM_LEAF][MAX_ATM_STR_LEN + 1],
       szLocalAddress[MAX_ATM_STR_LEN + 1],
       szPort[3];
SOCKET sLeafSock[MAX_ATM_LEAF];

// Module: usage
//
// Description:
//    Print usage information
//
void usage(char *progname)
{
    printf("usage: %s [-s]\n", progname);
    printf("     -s       Act as root\n");
    printf("     -l:str   Leaf address to invite (38 chars)\n");
    printf("              May be specified multiple times\n");
    printf("     -i:str   Local interface to bind to (38 chars)\n");
    printf("     -p:xx    Port number (2 hex chars)\n");
```

```
    printf("     -n:int   Number of packets to send\n");
    ExitProcess(1);
}

// Module: ValidateArgs
//
// Description:
//    Parse command line arguments
//
void ValidateArgs(int argc, char **argv)
{
    int      i;

    memset(szLeafAddresses, 0,
        MAX_ATM_LEAF * (MAX_ATM_STR_LEN + 1));
    memset(szPort, 0, sizeof(szPort));

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 's':     // Server
                    bServer = TRUE;
                    break;
                case 'l':     // Leaf address
                    if (strlen(argv[i]) > 3)
                    {
                        strncpy(szLeafAddresses[dwAddrCount++],
                            &argv[i][3], MAX_ATM_STR_LEN - 2);
                    }
                    break;

                case 'i':     // Local interface
                    if (strlen(argv[i]) > 3)
                    {
                        strncpy(szLocalAddress, &argv[i][3],
                            MAX_ATM_STR_LEN - 2);
                        bLocalAddress = TRUE;
                    }
                    break;
                case 'p':     // Port address to use
                    if (strlen(argv[i]) > 3)
                        strncpy(szPort, &argv[i][3], 2);
                    break;
                case 'n':     // Number of packets to send
                    if (strlen(argv[i]) > 3)
                        dwDataCount = atoi(&argv[i][3]);
                    break;
                default:
                    usage(argv[0]);
                    break;
```

```
            }
        }
    }
    return;
}

//
// Function: Server
//
// Description:
//    Bind to the local interface, and then invite each leaf
//    address that was specified on the command line.
//    Once each connection is made, send some data.
//
void Server(SOCKET s, WSAPROTOCOL_INFO *lpSocketProtocol)
{
    // Server routine
    //
    SOCKADDR_ATM  atmleaf, atmroot;
    WSABUF        wsasend;
    char          sendbuf[BUFSIZE],
                  szAddr[BUFSIZE];
    DWORD         dwBytesSent,
                  dwAddrLen = BUFSIZE,
                  dwNumInterfaces,
                  i;
    int           ret;

    // If no specified local interface is given, pick the
    // first one
    //
    memset(&atmroot, 0, sizeof(SOCKADDR_ATM));
    if (!bLocalAddress)
    {
        dwNumInterfaces = GetNumATMInterfaces(s);
        GetATMAddress(s, 0, &atmroot.satm_number);
    }
    else
        AtoH(&atmroot.satm_number.Addr[0], szLocalAddress,
            ATM_ADDR_SIZE - 1);
    //
    // Set the port number in the address structure
    //
    AtoH(&atmroot.satm_number.Addr[ATM_ADDR_SIZE-1], szPort, 1);
    //
    // Fill in the rest of the SOCKADDR_ATM structure
    //
    atmroot.satm_family                 = AF_ATM;
    atmroot.satm_number.AddressType     = ATM_NSAP;
    atmroot.satm_number.NumofDigits     = ATM_ADDR_SIZE;
    atmroot.satm_blli.Layer2Protocol    = SAP_FIELD_ANY;
    atmroot.satm_blli.Layer3Protocol    = SAP_FIELD_ABSENT;
    atmroot.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
```

```
    //
    // Print out what we're binding to, and bind
    //
    if (WSAAddressToString((LPSOCKADDR)&atmroot,
        sizeof(atmroot), lpSocketProtocol, szAddr,
        &dwAddrLen))
    {
        printf("WSAAddressToString failed: %d\n",
            WSAGetLastError());
    }
    printf("Binding to: <%s>\n", szAddr);

    if (bind(s, (SOCKADDR *)&atmroot,
        sizeof(SOCKADDR_ATM)) == SOCKET_ERROR)
    {
        printf("bind() failed: %d\n", WSAGetLastError());
        return;
    }
    // Invite each leaf
    //
    for(i = 0; i < dwAddrCount; i++)
    {

        // Fill in the SOCKADDR_ATM structure for each leaf
        //
        memset(&atmleaf, 0, sizeof(SOCKADDR_ATM));
        AtoH(&atmleaf.satm_number.Addr[0], szLeafAddresses[i],
            ATM_ADDR_SIZE - 1);
        AtoH(&atmleaf.satm_number.Addr[ATM_ADDR_SIZE - 1],
            szPort, 1);

        atmleaf.satm_family                 = AF_ATM;
        atmleaf.satm_number.AddressType     = ATM_NSAP;
        atmleaf.satm_number.NumofDigits     = ATM_ADDR_SIZE;
        atmleaf.satm_blli.Layer2Protocol    = SAP_FIELD_ANY;
        atmleaf.satm_blli.Layer3Protocol    = SAP_FIELD_ABSENT;
        atmleaf.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
        //
        // Print out client's address, and then invite it
        //
        if (WSAAddressToString((LPSOCKADDR)&atmleaf,
            sizeof(atmleaf), lpSocketProtocol, szAddr,
            &dwAddrLen))
        {
            printf("WSAAddressToString failed: %d\n",
                WSAGetLastError());
        }
        printf("[%02d] Inviting: <%s>\n", i, szAddr);

        if ((sLeafSock[i] = WSAJoinLeaf(s,
            (SOCKADDR *)&atmleaf, sizeof(SOCKADDR_ATM), NULL,
            NULL, NULL, NULL, JL_SENDER_ONLY))
            == INVALID_SOCKET)
```

```
        {
            printf("WSAJoinLeaf() failed: %d\n",
                WSAGetLastError());
            WSACleanup();
            return;
        }
    }
    // Note that the ATM protocol is a bit different from TCP.
    // When the WSAJoinLeaf (or connect) call completes, the
    // peer has not necessarily accepted the connection yet,
    // so immediately sending data will result in an error;
    // therefore, we wait for a short time.
    //
    printf("Press a key to start sending.");
    getchar();
    printf("\n");
    //
    // Now send some data to the group address, which will
    // be replicated to all clients
    //
    wsasend.buf = sendbuf;
    wsasend.len = 128;
    for(i = 0; i < dwDataCount; i++)
    {
        memset(sendbuf, 'a' + (i % 26), 128);
        ret = WSASend(s, &wsasend, 1, &dwBytesSent, 0, NULL,
            NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSASend() failed: %d\n", WSAGetLastError());
            break;
        }
        printf("[%02d] Wrote: %d bytes\n", i, dwBytesSent);
        Sleep(500);
    }

    for(i = 0; i < dwAddrCount; i++)
        closesocket(sLeafSock[i]);
    return;
}

//
// Function: Client
//
// Description:
//    First the client binds to the local interface (either one
//    specified on the command line or the first local ATM address).
//    Next it waits on an accept call for the root invitation. It
//    then waits to receive data.
//
void Client(SOCKET s, WSAPROTOCOL_INFO *lpSocketProtocol)
{
    SOCKET       sl;
```

```
    SOCKADDR_ATM atm_leaf,
                 atm_root;
    DWORD        dwNumInterfaces,
                 dwBytesRead,
                 dwAddrLen=BUFSIZE,
                 dwFlags,
                 i;
    WSABUF       wsarecv;
    char         recvbuf[BUFSIZE],
                 szAddr[BUFSIZE];

    int          iLen = sizeof(SOCKADDR_ATM),
                 ret;

    // Set up the local interface
    //
    memset(&atm_leaf, 0, sizeof(SOCKADDR_ATM));
    if (!bLocalAddress)
    {
        dwNumInterfaces = GetNumATMInterfaces(s);
        GetATMAddress(s, 0, &atm_leaf.satm_number);
    }
    else
        AtoH(&atm_leaf.satm_number.Addr[0], szLocalAddress,
            ATM_ADDR_SIZE-1);
    AtoH(&atm_leaf.satm_number.Addr[ATM_ADDR_SIZE - 1],
        szPort, 1);
    //
    // Fill in the SOCKADDR_ATM structure
    //
    atm_leaf.satm_family                = AF_ATM;
    atm_leaf.satm_number.AddressType    = ATM_NSAP;
    atm_leaf.satm_number.NumofDigits    = ATM_ADDR_SIZE;
    atm_leaf.satm_blli.Layer2Protocol   = SAP_FIELD_ANY;
    atm_leaf.satm_blli.Layer3Protocol   = SAP_FIELD_ABSENT;
    atm_leaf.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
    //
    // Print the address we're binding to, and bind
    //
    if (WSAAddressToString((LPSOCKADDR)&atm_leaf,
        sizeof(atm_leaf), lpSocketProtocol, szAddr,
        &dwAddrLen))
    {
        printf("WSAAddressToString failed: %d\n",
            WSAGetLastError());
    }
    printf("Binding to: <%s>\n", szAddr);

    if (bind(s, (SOCKADDR *)&atm_leaf, sizeof(SOCKADDR_ATM))
            == SOCKET_ERROR)
    {
        printf("bind() failed: %d\n", WSAGetLastError());
        return;
```

```
    }
    listen(s, 1);
    //
    // Wait for the invitation
    //
    memset(&atm_root, 0, sizeof(SOCKADDR_ATM));
    if ((sl = WSAAccept(s, (SOCKADDR *)&atm_root, &iLen, NULL,
        0)) == INVALID_SOCKET)
    {
        printf("WSAAccept() failed: %d\n", WSAGetLastError());
        return;
    }
    printf("Received a connection!\n");

    // Receive some data
    //
    wsarecv.buf = recvbuf;
    for(i = 0; i < dwDataCount; i++)
    {
        dwFlags = 0;
        wsarecv.len = BUFSIZE;
        ret = WSARecv(sl, &wsarecv, 1, &dwBytesRead, &dwFlags,
            NULL, NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSARecv() failed: %d\n", WSAGetLastError());
            break;
        }
        if (dwBytesRead == 0)
            break;
        recvbuf[dwBytesRead] = 0;
        printf("[%02d] READ %d bytes: '%s'\n", i, dwBytesRead,
            recvbuf);
    }
    closesocket(sl);
    return;
}

//
// Function: main
//
// Description:
//    This function loads Winsock library, parses command line
//    arguments, creates the appropriate socket (with the right
//    root or leaf flags), and starts the client or the server
//    functions, depending on the specified flags
//
int main(int argc, char **argv)
{
    WSADATA             wsd;
    SOCKET              s;
    WSAPROTOCOL_INFO    lpSocketProtocol;
    DWORD               dwFlags;
```

```
    ValidateArgs(argc, argv);
    //
    // Load the Winsock library
    //
    if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
    {
        printf("WSAStartup failed\n");
        return -1;
    }
    // Find an ATM-capable protocol
    //
    if (FindProtocol(&lpSocketProtocol) == FALSE)
    {
        printf("Unable to find ATM protocol entry!\n");
        return -1;
    }
    // Create the socket using the appropriate root or leaf flags
    //
    if (bServer)
        dwFlags = WSA_FLAG_OVERLAPPED
                  | WSA_FLAG_MULTIPOINT_C_ROOT
                  | WSA_FLAG_MULTIPOINT_D_ROOT;
    else
        dwFlags = WSA_FLAG_OVERLAPPED
                  | WSA_FLAG_MULTIPOINT_C_LEAF
                  | WSA_FLAG_MULTIPOINT_D_LEAF;

    if ((s = WSASocket(FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
            FROM_PROTOCOL_INFO, &lpSocketProtocol, 0,
            dwFlags)) == INVALID_SOCKET)
    {
        printf("socket failed with: %d\n", WSAGetLastError());
        WSACleanup();
        return -1;
    }
    // Start the correct driver, depending on which flags were
    // supplied on the command line
    //
    if (bServer)
    {
        Server(s, &lpSocketProtocol);
    }
    else
    {
        Client(s, &lpSocketProtocol);
    }
    closesocket(s);

    WSACleanup();
    return 0;
}
```

Common Winsock Options

Three socket options pertain to both the Winsock 1 and Winsock 2 implementations of Winsock: *IP_MULTICAST_TTL*, *IP_MULTICAST_IF*, and *IP_MULTICAST_LOOP*. These options are commonly associated with the Winsock 1 options for joining and leaving multicast groups; however, they can be equally useful with the Winsock 2 multicast functions. Of course, all three of the socket options pertain only to IP multicasting.

IP_MULTICAST_TTL

This option sets the TTL value for multicast data. By default, the TTL value is 1, which means that multicast data is dropped by the first router that encounters the data and only multicast members on the same network will receive the data. If you increase the TTL, the multicast data can cross multiple routers—equal to the value set for the TTL. Whenever a router receives a packet and determines that it should forward the packet to its adjoining networks, the TTL value is decremented by 1. If the newly decremented value is 0, the router drops the packet because its lifetime has expired. A multicast router does not forward multicast datagrams with destination addresses between 224.0.0.0 and 224.0.0.255 inclusive, regardless of their TTLs. This particular range of addresses is reserved for the use of routing protocols and other low-level topology discovery and maintenance protocols, such as gateway discovery and group membership reporting.

When you call *setsockopt*, the *level* parameter is *IPPROTO_IP*, the *optname* parameter is *IP_MULTICAST_TTL*, and the *optval* parameter is an integer that indicates the new TTL value. The following code snippet illustrates how to set the TTL value.

```
int    optval;

optval = 8;
if (setsockopt(s, IPPROTO_IP, IP_MULTICAST_TTL, (char *)&optval,
        sizeof(int)) == SOCKET_ERROR)
{
    // Error
}
```

In addition to this socket option, the option *SIO_MULTICAST_SCOPE* exists for use with the *WSAIoctl* or *ioctlsocket* function that performs the same operation on a socket.

A TTL parameter is not necessary in ATM multicasting because sending data on an ATM is one way only and all recipients are known. Because the control plane is rooted, the c_root node must explicitly invite each leaf to join; this means that you don't need to limit the scope of a data transmission so that data isn't duplicated on networks where there might not be any multicast participants.

IP_MULTICAST_IF

This option sets the IP interface that multicast data is sent from when you send multicast data. Under normal (nonmulticast) circumstances, the routing table is the sole determinant of the interface a datagram goes out on. The system determines which interface is best suited for a particular datagram and its destination. However, since multicast addresses can be used by anyone, the routing table isn't sufficient. The programmer must possess some knowledge of where the multicast data should go. Of course, this is only an issue if the machine on which multicast data is being sent is connected to multiple networks via network cards. For this option, the *optval* parameter is the address of the local interface on which to send multicast data. The following code illustrates this.

```
DWORD    dwInterface;

dwInterface = inet_addr("129.121.32.19");
if (setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, (char *)&dwInterface,
    sizeof(DWORD)) == SOCKET_ERROR)
{
    // Error
}
```

In the above example, we see that we are setting the local interface to 129.121.32.19. Any multicast data sent on socket *s* goes out on the network interface on which that IP address is assigned.

Again, ATM does not require a separate socket option to set an interface. The c_root can explicitly bind to a particular interface before calling *WSAJoinLeaf*. Likewise, the client must bind to a specific ATM interface in order to wait for an invitation with *accept* or *WSAAccept*.

IP_MULTICAST_LOOP

The last socket option determines whether your application receives its own multicast data. If your application joins a multicast group and sends data to that group, your application will receive that data. If you have a *recvfrom* call pending when the data was sent, the call will return with a copy of that data. Note that it is not necessary for an application to join a multicast group in order for the application to send data to the multicast group. You are required to join the group only if you want to receive data destined for that group. The *IP_MULTICAST_LOOP* socket option is designed to turn off this echoing of data back to the local interface. With this option, you pass an integer value as the *optval* parameter, which is a simply Boolean value indicating whether to enable or disable multicast loopback. The following code example shows this.

```
int    optval;

optval = 0;     // Disable loopback
if (setsockopt(s, IPPROTO_IP, IP_MULTICAST_LOOP, (char *)&optval,
    sizeof(int)) == SOCKET_ERROR)
{
    // Error
}
```

Additionally, an accompanying ioctl command used with *WSAIoctl* or *ioctlsocket—SIO_MULTIPOINT_LOOPBACK*—performs the same function. Unfortunately, this socket option is not implemented on Windows 95, Windows 98, or Windows NT 4, and by default the loopback is enabled. If you attempt to disable loopback by use of this socket option, you will receive the error *WSAENOPROTOOPT*.

By definition, an ATM root node, which is the only member allowed to send data to the multicast group, will not receive its own data because the root socket is not a leaf socket. Only leaf sockets can receive data in ATM multicasting. The same process that creates the c_root can create a separate c_leaf node that the c_root can invite; however, this is not a true loopback.

One Limitation of Dial-Up Networking Multicasting

There is one limitation to be aware of when attempting to send or receive multicast data on a Remote Access Service (RAS), or dial-up, interface. This limitation actually lies within the server you dial into. The majority of Windows-based dial-in servers run Windows NT 4, which does not have an IGMP proxy. This means that any group membership requests you make won't make it off of the Windows NT 4 server. As a result, your application cannot join any multicast group and therefore cannot send or receive multicast data. The RAS server for Windows 2000 does contain an IGMP proxy, although it is not enabled by default. However, once it is enabled, dial-in clients will be able to join groups as well as send and receive multicast data.

Conclusion

Multicasting offers a number of advantages for applications that need to communicate with multiple endpoints without the overhead of broadcasting. In this chapter, we defined multicasting and presented the different multicasting models. We then discussed how IP multicasting and ATM point-to-multipoint communication apply to these models. Finally, we covered how the Winsock API supports multicasting by presenting both the Winsock 1 method, which uses socket options, and the newer Winsock 2 method, which uses the *WSAJoinLeaf* function.

Chapter 12

Generic Quality of Service

With the wide variety of multimedia applications available today, as well as the popularity of the Internet, many networks are becoming saturated with the traffic of these bandwidth-hungry applications. This is especially a problem on shared media networks—such as Ethernet—because all traffic is treated equally and a single application can flood the network. Quality of Service (QOS) is a set of components that allows the differentiation and preferential treatment of data on the network. A QOS-enabled network can be configured to offer programmers the following capabilities:

- Prevent nonadaptive protocols (such as UDP) from abusing network resources
- Partition resources between "best-effort" traffic and higher-priority or lower-priority traffic
- Reserve resources for entitled users
- Prioritize access to resources based on the user

Generic Quality of Service (GQOS) is Microsoft's implementation of QOS. Currently Microsoft supplies a QOS-enabled TCP/IP and UDP/IP provider that is available on Windows 98 and Windows 2000. Note that the ATM protocol is capable of QOS because it is available natively in ATM.

This chapter covers QOS and how it is implemented on Win32 platforms. We will begin with a discussion of the various components that need to be in place to allow preferential treatment of network traffic. Then we'll examine how the Winsock interface exposes the ability to write a network application that can take advantage of these components for time- and bandwidth-critical applications. The majority of this chapter is dedicated to QOS on IP networks. At the end of this chapter, we will discuss QOS on ATM networks, which is slightly different from QOS on IP networks.

> NOTE
>
> Throughout the chapter, we will refer to Quality of Service as QOS. You can assume that we are discussing Microsoft's implementation of QOS.

Background

QOS requires three components to make it work:

- Devices on the network—such as routers and switches—that are aware of this differentiation of services
- Local workstations that can prioritize traffic that they place on the network
- The policy component: who is allowed to use the available bandwidth and how much they are allowed to use

However, before we begin discussing these components, we need to look at the Resource Reservation Protocol, which is the signaling protocol used between QOS senders and QOS receivers. RSVP plays a major role in QOS and the integration of the three major components of QOS.

Resource Reservation Protocol

Resource Reservation Protocol (RSVP) is the glue that binds the network, application, and policy components into one cohesive unit. RSVP carries resource reservation requests through the network, which can be composed of different media. RSVP propagates a user's QOS requests to all RSVP-aware network devices along the data path, allowing resources to be reserved from all RSVP-enabled devices. As a result, the network nodes can indicate whether the network can meet the desired levels of service.

The RSVP protocol reserves network resources by establishing *flows* end to end through the network. A flow is a network path associated with one or more senders, one or more receivers, and a specific level of QOS. A sending host wanting to send data that requires a specific level of QOS issues a PATH message toward the intended recipient or recipients. This PATH message contains the bandwidth requirements. The relevant parameters are propagated along the path to the intended recipients.

A receiving host that is interested in this data reserves the resources for the flow (and the entire path from the sender) by sending a RESV (reserve) message back toward the sender. As this occurs, intermediate RSVP-enabled devices decide whether they can accommodate the requested bandwidth requirements and ensure that the user who is requesting resources actually has the permission to do so. If the requested bandwidth is available and the user's policy settings indicate the user has the right to the request, each intermediate RSVP-enabled device commits the resources and propagates the RESV message back toward the sender.

When the sender receives the RESV message, QOS data can begin to flow. Periodically, each endpoint within the flow sends out PATH and RESV messages to reaffirm the reservation and to provide network information in case the levels of available bandwidth change. Also, by periodically refreshing PATH and RESV messages, the RSVP protocol remains dynamic. In the event that a better (for example, faster) route becomes available, these refresh messages can discover a new route. When we discuss QOS from Winsock later in this chapter, we'll return to RSVP and how the Winsock API calls invoke it.

Be aware of one important aspect of the session setup and RSVP: it is a one-way reservation. This is the case even if the application requests bandwidth requirements for both sending and receiving. One session is initiated for the sending requirements and another session is started for the receiving requirements. Later in this chapter, we will discuss the criteria required for initiating an RSVP session.

Network Components

In order for end-to-end QOS to work, the network devices between the two endpoints must also be able to differentiate traffic priorities. This way they can route traffic in a manner that satisfies the QOS guarantee that an application received. Additionally, these network devices must be able to determine whether enough bandwidth is available on the network when an application requests bandwidth. To support these requirements, the following components have been created:

- 802.1p A standard for prioritizing packets in a subnet by setting 3 bits within the MAC header of packets
- IP Precedence A method to establish priority for IP packets

- Layer 2 signaling A mechanism for mapping RSVP objects to native WAN QOS components in OSI Layer 2 of a network
- Subnet Bandwidth Manager (SBM) A component that manages shared media network bandwidth
- Resource Reservation Protocol (RSVP) A protocol that carries QOS requests and information to QOS-aware network devices along the path between a sender and one or more receivers

802.1p

A major part of enforcing QOS provisions and avoiding treating all packets equally lies on hubs and switches within the network. Hubs and switches lie within Layer 2 of the OSI reference model and as a result are aware of fields only within the media access control (MAC) header at the beginning of each packet.

802.1p is a standard that prioritizes network packets by setting a 3-bit precedence value in the MAC header. When a subnet becomes congested on non-802.1p networks, switches and routers are unable to keep up with the amount of traffic and a delay is introduced. On the other hand, switches and routers on 802.1p networks can begin prioritizing incoming traffic based on the precedence bits and give the higher-priority packets preferential treatment.

Implementing 802.1p for QOS requires specialized hardware capable of recognizing this 3-bit field. Network interface cards (NIC), network drivers, and network switches all must be 802.1p-aware.

IP Precedence

IP Precedence is a method of specifying precedence values at a higher level than 802.1p. This method allows packets passing through OSI Layer 3 devices—such as routers—to have their relative priorities differentiated. IP Precedence is implemented by using the type of service (TOS) field within the IP header to establish varying levels of priority. Based on these bits, routers can establish priority queues to service the different priority levels, which means higher priority traffic receives better service from routers.

As with 802.1p, in order for IP Precedence to work, all Layer 3 devices on the network must be able to understand the significance of the IP Precedence bits and handle traffic accordingly.

Layer 2 signaling

Layer 2 signaling is necessary when traffic traverses a wide area network (WAN). Typically a WAN links several networks over a variety of communications hardware. Thus, a WAN can manipulate Layer 1, Layer 2, and Layer 3 information as data is transmitted. To guarantee end-to-end QOS, the WAN link must understand the prioritization of QOS traffic. To accomplish this understanding, QOS provides a method for mapping RSVP and other QOS parameters to the WAN's native underlying Layer 2 signaling method—the means by which WAN technologies implement their own native QOS.

Subnet Bandwidth Manager

The Subnet Bandwidth Manager (SBM) manages the resources on a given shared media network, such as Ethernet. The SBM is also responsible for handling policy-based admission control for QOS applications. An SBM is necessary on shared media networks because when an endpoint requests QOS for an application, each network device admits or rejects the request based on the allocation of the network device's private resources. The network devices are not aware of the available resources on the shared media. The SBM solves this problem by becoming a broker for these devices. The SBM is also closely tied to the Admission Control Service (ACS) that is a part of the policy component. The SBM must check to ensure that an application (or user) requesting bandwidth has the privileges to do so. Note that the SBM for a network can be a host running Windows 2000 Server.

Application Components

You now have a good idea of network requirements for supporting QOS. We must consider how the local system prioritizes data based on the QOS levels that an application has requested. For the local system to support QOS,

the following components are necessary:

- GQOS service provider A service provider that invokes other QOS components
- Traffic Control module The module that controls the traffic leaving the computer (This module includes the Generic Packet Classifier, the Packet Scheduler, and the Packet Shaper.)
- Resource Reservation Protocol (RSVP) The protocol that is invoked by the GQOS service provider and that carries the reservation request across the network
- GQOS API The programmatic interface to GQOS, such as Winsock
- Traffic Control (TC) API The programmatic interface to the Traffic Control components regulating traffic on the local host

GQOS service provider

The GQOS service provider is the GQOS component that invokes nearly all resulting QOS facilities. The GQOS service provider initiates Traffic Control functionality (if appropriate) and implements, maintains, and handles RSVP signaling for all GQOS functionality.

In order to find the QOS-enabled service provider(s) on your host, you can query the provider catalog with *WSAEnumProtocols*. The flag to check whether a provider is QOS-enabled is within the *WSAPROTOCOL_INFO* structure returned from *WSAEnumProtocols*. The field of interest is *dwServiceFlags1*, and the flag to check for is *XP1_QOS_SUPPORTED*. For more information about *WSAEnumProtocols*, consult Chapter 5.

Traffic Control module

Traffic Control (TC) plays a significant and central role in QOS. Within TC, packets are prioritized both within and outside the network node on which TC is enabled. The effects of this preferential treatment of packets as they flow through the system and through the network reach across the entire network and therefore directly affect QOS characteristics. The TC module is implemented through three modules: the Generic Packet Classifier, the Packet Scheduler, and the Packet Shaper.

Generic Packet Classifier

The duty of the Generic Packet Classifier (GPC) is to classify and prioritize packets within network components. The GPC performs this prioritization for activities such as CPU time or transmission onto the network.

The GPC accomplishes this prioritization by creating lookup tables and classification services within the network stack. This becomes the first step in the prioritization process for network traffic.

Packet Scheduler

Packet scheduling controls the way data transmission is performed, which is a key function of QOS. The Packet Scheduler is the traffic control module that regulates how much data an application, or flow, is allowed, essentially enforcing QOS parameters set for a particular flow.

The Packet Scheduler takes the prioritization scheme provided by the GPC and provides different levels of service to the various priority levels. For example, data that has been classified by the GPC as high priority receives preferential treatment within the Packet Scheduler.

Packet Shaper

The purpose of the Packet Shaper is to regulate the transmission of data from data flows onto the network. Most applications read and write data in bursts; however, many QOS applications need a particular data rate for sent data. Therefore, the Packet Shaper schedules the transmission of data over a period of time, smoothing out network usage and resulting in a more evenly loaded network.

Traffic Control API

The Traffic Control API is the interface to the components that regulate network traffic on the local host. This includes methods for manipulating the Generic Packet Classifier, the Packet Scheduler, and the Packet Shaper. Some of the Traffic Control functions are implicitly invoked through calls made to Winsock GQOS-enabled functions that are serviced by the GQOS Service Provider. However, applications that need to manipulate the Traffic Control components directly can do so with the Traffic Control API functions. These functions are beyond the scope of this book. (Consult the Platform SDK for more information.) We will cover the Winsock GQOS API later in this chapter.

Policy Components

Policy, the third and final component of GQOS, controls the allocation of resources to QOS-enabled applications. Policy components are of most interest to system administrators who want to control the allocation of resources based on users or on the class of application requesting bandwidth. Policy components include the following:

- Admission Control Service (ACS) A Windows 2000 Server service that intercepts RSVP PATH and RESV messages to control access of QOS-enabled clients to the various levels of guarantees offered through QOS
- Local Policy Module (LPM) Provides resource-access decisions based on policies configured through ACS for the SBM
- Policy Element (PE) Resides on the client and provides authentication information to facilitate reservation requests

Admission Control Service

The Admission Control Service (ACS) regulates network usage for QOS-enabled applications. This is done through the RSVP protocol. The ACS intercepts both PATH and RESV messages in order to verify that the requesting application has sufficient privileges. Once an RSVP message is intercepted, it is passed to the Local Policy Module (LPM), which performs the actual authentication.

The ACS resides on a Windows 2000 machine and can be configured by the system administrator, who can set resource limits on users, applications, or groups.

Local Policy Module

The Local Policy Module (LPM) is closely related to the ACS in that the ACS intercepts RSVP messages, inserts user information, and passes them to the LPM. At this point, the LPM looks the user up in the Active Directory to verify policy information. If network resources are available (as determined by the SBM), and if the authentication check succeeds, the RSVP message intercepted by the ACS is sent to the next hop. Of course, if the user does not have the necessary permissions to request a certain level of QOS, an error indicating this is generated and returned within the RSVP message.

> NOTE
>
> For policy checks to succeed, users must be part of a Windows 2000 domain.

Policy Element

This component actually contains the policy information that the Local Policy Module requests. These data structures are not covered in this book because they deal mainly with administration of network resources, which is not the focus of the book.

QOS and Winsock

In the previous section, we discussed the various components required for the success of an end-to-end QOS network. Now we'll turn our attention to Winsock 2, which is the API you use to programmatically access QOS from an application. First we'll take a look at the top-level QOS structures needed by the majority of Winsock calls. Next we'll cover the Winsock functions capable of invoking QOS on a socket, as well as how to terminate QOS once it has been enabled on a socket. The last thing we'll do is cover the provider-specific objects that can be used to affect the behavior of—or return information from—the QOS service provider.

It might seem a bit out of order to jump from a discussion of the major QOS structures to QOS functions and then back to provider-specific structures. However, we want to give a thorough, high-level overview of how the major structures interact with the Winsock API calls before delving into the gory details of the provider-specific options.

QOS Structures

The central structure in QOS programming is the *QOS* structure. This structure consists of

- A *FLOWSPEC* structure used to describe the QOS levels that your application will use for sending data
- A *FLOWSPEC* structure used to describe the QOS levels that your application will use to receive data
- A service provider-specific buffer to allow the specification of provider-specific QOS characteristics (We will discuss these characteristics in the provider-specific section.)

QOS

The *QOS* structure specifies the QOS parameters for both sending and receiving traffic. It is defined as

```
typedef struct _QualityOfService
{
    FLOWSPEC   SendingFlowspec;
    FLOWSPEC   ReceivingFlowspec;
    WSABUF     ProviderSpecific;
} QOS, FAR * LPQOS;
```

The *FLOWSPEC* structures define the traffic characteristics and requirements for each traffic direction, while the *ProviderSpecific* field is used to return information and to change the behavior of QOS. These provider-specific options are covered in detail a little later in this chapter.

FLOWSPEC

FLOWSPEC is the basic structure that describes a single flow. Remember that a flow describes data traveling in a single direction. The structure is defined as

```
typedef struct _flowspec
{
    ULONG        TokenRate;
    ULONG        TokenBucketSize;
    ULONG        PeakBandwidth;
    ULONG        Latency;
    ULONG        DelayVariation;
    SERVICETYPE  ServiceType;
    ULONG        MaxSduSize;
    ULONG        MinimumPolicedSize;
} FLOWSPEC, *PFLOWSPEC, FAR *LPFLOWSPEC;
```

Let's take a look at the meaning of each of the *FLOWSPEC* structure fields.

TokenRate

The *TokenRate* field specifies the rate of transmission for data that is given in bytes per second. An application can transmit data at this rate, but if for some reason it transmits data at a lower rate, the application can accrue extra tokens so that more data can be transmitted later. However, the number of tokens that an application can accrue is bound by *PeakBandwidth*. This accumulation of token credits itself is limited by the *TokenBucketSize* field. By limiting the total number of tokens, we avoid a situation of inactive flows that have accrued many tokens, which could lead to flooding the available bandwidth. Because flows can accrue transmission credits over time (at their *TokenRate* value) *only* up to the maximum of their *TokenBucketSize*, and because they are limited in "burst transmissions" to their *PeakBandwidth*, traffic control and network-device resource integrity are maintained. Traffic control is maintained because flows cannot send too much data at once, and network-device resource integrity is maintained because such devices are spared high-traffic bursts.

Because of these limitations, an application can transmit only when sufficient credits have accrued. If the required number of credits are not available, the application must either wait until sufficient credits have accrued to send the data or discard the data altogether. The Traffic Control module determines what happens to data queued too long without being sent. Therefore, applications should take care to base their TokenRate requests on reasonable amounts.

If an application does not require scheduling of transmission rates, this field can be set to *QOS_NOT_SPECIFIED* (-1).

TokenBucketSize

As we discussed earlier, the *TokenBucketSize* field limits the number of credits that can accrue for a given flow. For example, video applications would set this field to the frame size being transmitted since it is desirable to have single video frames being transmitted at a time. Applications requiring a constant data rate should set this field to allow for some variation. Like the *TokenRate* field, *TokenBucketSize* is expressed in bytes per second.

PeakBandwidth

PeakBandwidth specifies the maximum amount of data transmitted in a given period of time. In effect, this value specifies the maximum amount of burst data. This is an important value because it prevents applications that have accrued a significant number of transmission tokens from flooding the network all at once. *PeakBandwidth* is expressed in bytes per second.

Latency

The *Latency* field specifies the maximum acceptable delay between the transmission of a bit and its receipt by the

intended recipient or recipients. How this value is actually interpreted depends on the level of service requested in the *ServiceType* field. *Latency* is expressed in microseconds.

DelayVariation

DelayVariation specifies the difference between the minimum and maximum delay that a packet can experience. Typically an application uses this value to determine the amount of buffer space required to receive the data and still maintain the original data transmission pattern. *DelayVariation* is expressed in microseconds.

ServiceType

The *ServiceType* field specifies the level of service required by the data flow. The following service types can be specified:

- *SERVICETYPE_NOTRAFFIC* indicates that no data is being transmitted in this direction.
- *SERVICETYPE_BESTEFFORT* indicates that the parameters specified in *FLOWSPEC* are guidelines and that the system will make a reasonable effort to maintain that service level; however, there are no guarantees of packet delivery.
- *SERVICETYPE_CONTROLLEDLOAD* indicates that data transmission will closely approximate transmission quality provided by best-effort service on a network with nonloaded traffic conditions. This really breaks down into two conditions. First, packet loss will approximate the normal error rate of the transmission medium; and second, transmission delay will not greatly exceed the minimum delay experienced by delivered packets.
- *SERVICETYPE_GUARANTEED* guarantees data transmission at the rate specified by the *TokenRate* field over the lifetime of the connection. However, if the actual data transmission rate exceeds *TokenRate*, data might be delayed or discarded (depending on how Traffic Control is configured). Additionally, if *TokenRate* is not exceeded, *Latency* is also guaranteed.

In addition to these four service types, several other flags provide information that can be returned to an application. These informational flags can be ORed with any valid *ServiceType* flag. Table 12-1 lists these information flags.

Table 12-1. *Service type modifier flags*

Value	*Meaning*
SERVICETYPE_NETWORK_UNAVAILABLE	Indicates a loss of service in either the sending or the receiving direction.
SERVICETYPE_GENERAL_INFORMATION	Indicates that all service types are supported for a flow.
SERVICETYPE_NOCHANGE	Indicates that there is no change in the requested QOS service level. This flag can be returned from a Winsock call or an application can specify this flag when renegotiating QOS to indicate no change in the QOS levels for the given direction.
SERVICE_IMMEDIATE_TRAFFIC_CONTROL	An application can use this flag to indicate to the system to immediately invoke Traffic Control instead of sending best effort until a RESV message arrives.
SERVICE_NO_TRAFFIC_CONTROL	This flag can be ORed with other *ServiceType* flags to disable traffic control altogether.
SERVICE_NO_QOS_SIGNALING	This flag can be used with the immediate traffic control flag above to prevent any RSVP signaling messages from being sent. Local traffic control will be invoked, but no RSVP Path messages will be sent. This flag can also be used in conjunction with a receiving *FLOWSPEC* structure to suppress the automatic generation of a RESV message. The application receives notification that a PATH message has arrived and

	then needs to alter the QOS by issuing *WSAIoctl*(*SIO_SET_QOS*) to unset this flag and thereby cause RESV messages to go out.

MaxSduSize

The *MaxSduSize* field indicates the maximum packet size for data transmitted in the given flow. *MaxSduSize* is expressed in bytes.

MinimumPolicedSize

The *MinimumPolicedSize* field indicates the minimum packet size that can be transmitted in the given flow. *MinimumPolicedSize* is expressed in bytes.

QOS-Invoking Functions

Let's say you want your application to make a request on the network for certain bandwidth requirements. Four functions initiate the process. Once an RSVP session has begun, an application can register for *FD_QOS* events. QOS status information and error codes are conveyed to applications as *FD_QOS* events. Applications can register to receive these events in the usual way: by including the *FD_QOS* flag in the event field of either the *WSAAsyncSelect* or the *WSAEventSelect* function.

The *FD_QOS* notification is especially relevant if a connection is established with *FLOWSPEC* structures that specify default values (*QOS_NOT_SPECIFIED*). Once the application has made the request for QOS, the underlying provider will periodically update the *FLOWSPEC* structure to indicate current network conditions and will notify the application by posting an *FD_QOS* event. With this information, applications can request or modify QOS levels to reflect the amount of available bandwidth. Keep in mind that the updated information is an indication of only the locally available bandwidth and does not necessarily indicate the end-to-end bandwidth.

Once a flow is established, available network bandwidth might change or a single party taking part in an established flow might decide to change the requested QOS service level. A renegotiation of allocated resources causes an *FD_QOS* event to be generated to indicate the change to the application. At this point, the application should call *SIO_GET_QOS* to obtain the new resource levels. We'll revisit QOS event signaling and status information in the section on programming QOS later in this chapter.

WSAConnect

A client uses the *WSAConnect* function to initiate a unicast QOS connection to a server. The requested QOS values are passed as the *lpSQOS* parameters. Currently group QOS is not supported or implemented; a null value should be passed for *lpGQOS*.

```
int WSAConnect (
    SOCKET s,
    const struct sockaddr FAR *name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS
);
```

The *WSAConnect* call can be used with connection-oriented or connectionless sockets. With a connection-oriented socket, this function establishes the connection and also generates the appropriate PATH and/or RESV messages. For connectionless sockets, you must associate an endpoint's address with the socket so that the service provider

knows where to send PATH and RESV messages. The caveat with using *WSAConnect* on a connectionless socket is that only data sent to that destination address will be shaped by the system according to the QOS levels associated with that socket. In other words, if *WSAConnect* is used to associate an endpoint on a connectionless socket, data can be transferred only between those two endpoints for the lifetime of the socket. If you need to send data with QOS guarantees to multiple endpoints, use *WSAIoctl* and *SIO_SET_QOS* to specify each new endpoint.

WSAAccept

The *WSAAccept* function accepts a client connection that can be QOS-enabled. The prototype for the function is as follows:

```
SOCKET WSAAccept(
    SOCKET s,
    struct sockaddr FAR *addr,
    LPINT  addrlen,
    LPCONDITIONPROC lpfnCondition,
    DWORD  dwCallbackData
);
```

If you want to supply a conditional function, you must prototype it as

```
int CALLBACK ConditionalFunc(
    LPWSABUF lpCallerId,
    LPWSABUF lpCallerData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
    LPWSABUF lpCalleeId,
    LPWSABUF lpCalleeData,
    GROUP FAR *g,
    DWORD dwCallbackData
);
```

The unfortunate drawback is that the QOS service provider does not guarantee to return the actual QOS values that the client is requesting as the *lpSQOS* parameter, which means that to enable QOS on the client socket, *WSAIoctl* with *SIO_SET_QOS* must be called before or after *WSAAccept*. If QOS is set on the listening socket, those values will be copied over to the client socket by default.

In actuality, the condition function is not useful at all. With TCP, you cannot reject a client connection per se because by the time the conditional function has been called, the connection has already been established on the TCP level. Additionally, the QOS service provider will not pass valid QOS parameters into the conditional function even if a PATH message has already arrived. Basically, don't use the *WSAAccept* condition function.

> NOTE
>
> There is one issue to be aware of when using *WSAAccept* on Windows 98. If you do use a conditional function with *WSAAccept* and the *lpSQOS* parameter is not null, you must set QOS (using *SIO_SET_QOS*), or *WSAAccept* will fail.

WSAJoinLeaf

WSAJoinLeaf is used for multipoint communications. Chapter 11 discusses multicasting in great detail. The function is defined as

```
SOCKET WSAJoinLeaf(
    SOCKET s,
    const struct sockaddr FAR *name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
    DWORD dwFlags
);
```

For an application to join a multicast session, it must create a socket with the appropriate flags (*WSA_FLAG_MULTIPOINT_C_ROOT*, *WSA_FLAG_MULTIPOINT_C_LEAF*, *WSA_FLAG_MULTIPOINT_D_ROOT*, and *WSA_FLAG_MULTIPOINT_D_LEAF*). When the application sets up multipoint communications, it specifies QOS parameters in the *lpSQOS* parameter.

When you use *WSAJoinLeaf* to join IP multicast groups, the operation of joining a multicast group is separate from the QOS RSVP session setup. In fact, joining a multicast group is likely to succeed. The function returns without the reservation request completing. At some later time, you will receive an *FD_QOS* event that will notify you of either a success or a failure in allocating the requested resources.

Keep in mind the time-to-live (TTL) set on multicast data. If you plan on setting the TTL with either *SIO_MULTICAST_SCOPE* or *IP_MULTICAST_TTL*, the TTL must be set prior to calling *WSAJoinLeaf* or calling the *SIO_SET_QOS* ioctl command to set QOS on the socket. If the scope is set after the QOS is already set, the TTL will not take effect until QOS is renegotiated through *SIO_SET_QOS*. The TTL value set will also be carried by the RSVP request.

Setting the TTL before setting QOS on a socket is important because the multicast TTL set on the socket also affects the TTL of the RSVP messages, directly affecting how many networks your resource reservation request is propagated to. For example, if you want to set up several endpoints in an IP multicast group that spans 3 networks, you ideally would set the TTL to 3 so that the network traffic you generate is not propagated to networks beyond those interested in the data. If the TTL isn't set before *WSAJoinLeaf* is called, RSVP messages are sent out with a default TTL of 63, which results in the host attempting to reserve resources on far too many networks.

WSAIoctl

The *WSAIoctl* function with the ioctl option *SIO_SET_QOS* can be used either to request QOS for the first time on either a connected or an unconnected socket or to renegotiate QOS requirements after an initial QOS request. The one advantage to using *WSAIoctl* is that if the QOS request fails, more detailed error information is returned via the provider-specific information. Chapter 9 covers the *WSAIoctl* function and how it is called, along with *SIO_SET_QOS* and *SIO_GET_QOS*.

The *SIO_SET_QOS* option is used to set or modify QOS parameters on a socket. One feature of using *WSAIoctl* with *SIO_SET_QOS* is the ability to specify provider-specific objects to further refine the behavior of QOS. The next section is dedicated to covering all the provider-specific objects. In particular, an application using connectionless sockets that does not want to use *WSAConnect* can call *WSAIoctl* with *SIO_SET_QOS* and specify the destination address object in the provider-specific buffer to associate an endpoint so that an RSVP session can be established. When setting QOS parameters, pass the QOS structure as *lpvInBuffer*, with *cbInBuffer* indicating the amount of bytes passed in.

The *SIO_GET_QOS* option is used upon receipt of an *FD_QOS* event. When an application receives this event

notification, a call to *WSAIoctl* with *SIO_GET_QOS* should be made to investigate the reason for the event. As we mentioned earlier, the *FD_QOS* event can be generated because of a change in the available bandwidth on the network or by renegotiation by the peer. To obtain the QOS values for a socket, pass a sufficiently large buffer as *lpvOutBuffer*, with *cbOutBuffer* indicating the size. The input parameters can be *NULL* and 0. The one tricky thing when calling *SIO_GET_QOS* is passing a buffer large enough to hold the *QOS* structure, including the provider-specific objects. The *ProviderSpecific* field—a *WASBUF* structure—is within the *QOS* structure. If the *len* field is set to *QUERY_PS_SIZE* and the *buf* field is null, *len* will be updated with the necessary size upon return from *WSAIoctl*. Additionally, if the call fails because the buffer is too small, the *len* field will be updated with the correct size. Querying for the buffer size is supported only on Windows 2000. For Windows 98, you must always supply a large enough buffer—simply pick a large buffer size and stick with it.

Another ioctl command can be used with *WSAIoctl*: *SIO_CHK_QOS*. This command can be used to query for the six values described in Table 12-2. When you call this command, the *lpvInBuffer* parameter points to a *DWORD* that is set to one of the three flags. The *lpvOutBuffer* parameter should also point to a *DWORD*, and upon return, the value requested is returned. The most commonly used flag is *ALLOWED_TO_SEND_DATA*. This flag is used by senders who have initiated a PATH message but have not received any RESV messages indicating successful allocation of the QOS level. When senders use the *SIO_CHK_QOS* ioctl command with the *ALLOWED_TO_SEND_DATA* flag, the network is queried to see whether the best-effort traffic currently available is sufficient for sending the kind of data described in the *QOS* structure passed to a QOS-invoking function. For more details, take a look at the entry for this ioctl command in Chapter 9.

Table 12-2. SIO_CHK_QOS *flags*

SIO_CHK_QOS *Flag*	*Description*	*Return Value*
ALLOW_TO_SEND_DATA	Indicates whether sending data can begin immediately or whether the application should wait for a RESV message	*BOOL*
ABLE_TO_RECV_RSVP	Indicates to senders whether its interface is RSVP-enabled	*BOOL*
LINE_RATE	Returns the bandwidth capacity of the interface	*DWORD*
LOCAL_TRAFFIC_CONTROL	Returns whether Traffic Control is installed and available for use	*BOOL*
LOCAL_QOSABILITY	Returns whether QOS is available	*BOOL*
END_TO_END_QOSABILITY	Determines whether end-to-end QOS is available on the network	*BOOL*

The options listed in Table 12-2 that return *BOOL* values actually return 1 or 0 to indicate a yes or a no answer respectively. The last four options in the table can return the constant *INFO_NOT_AVAILABLE* if the system cannot currently obtain the answer.

Terminating QOS

In the previous section, you learned how to invoke QOS on a socket. Next we'll examine the termination of QOS guarantees. Each of the following events causes a termination of RSVP and Traffic Control processing for a socket:

- Closing a socket via the *closesocket* function
- Shutting down a socket via the *shutdown* function
- Calling *WSAConnect* with a null peer address
- Calling *WSAIoctl* and *SIO_SET_QOS* with the *SERVICETYPE_NOTRAFFIC* or the *SERVICE_TYPE_BESTEFFORT* service type

Except for the second item in the list, these events are self-explanatory. Remember that the *shutdown* function can signal the cessation of either sending or receiving data, which will result in the termination of the flow of data for only that direction. In other words, if *shutdown* is called with *SD_SEND*, QOS will still be in effect for data being received.

Provider-Specific Objects

The provider-specific objects covered in this section are passed as part of the *ProviderSpecific* field of the *QOS* structure. Either they return QOS information to your application via the *FD_QOS* event or you can pass them along with the other QOS parameters to a *WSAIoctl* call with the *SIO_SET_QOS* option to refine the behavior of QOS.

Every provider-specific object contains a *QOS_OBJECT_HDR* structure as its first member. This structure identifies the type of the provider-specific object. This is necessary because these provider objects are most commonly returned within the *QOS* structure after a call to *SIO_GET_QOS*. Using the *QOS_OBJECT_HDR*, your application can identify each object and decode its significance. The object header is defined as

```
typedef struct
{
    ULONG   ObjectType;
    ULONG   ObjectLength;
} QOS_OBJECT_HDR, *LPQOS_OBJECT_HDR;
```

ObjectType identifies the type of preset provider-specific object, while *ObjectLength* tells how long the entire object is, including the object header and the provider-specific object. An object type can be one of the flags listed in Table 12-3.

Table 12-3. *Object types*

Provider Object	*Object Structure*
QOS_OBJECT_PRIORITY	*QOS_PRIORITY*
QOS_OBJECT_SD_MODE	*QOS_SD_MODE*
QOS_OBJECT_TRAFFIC_CLASS	*QOS_TRAFFIC_CLASS*
QOS_OBJECT_DESTADDR	*QOS_DESTADDR*
QOS_OBJECT_SHAPER_QUEUE_DROP_MODE	*QOS_SHAPER_QUEUE_LIMIT_DROP_MODE*
QOS_OBJECT_SHAPER_QUEUE_LIMIT	*QOS_SHAPER_QUEUE_LIMIT*
RSVP_OBJECT_STATUS_INFO	*RSVP_STATUS_INFO*
RSVP_OBJECT_RESERVE_INFO	*RSVP_RESERVE_INFO*
RSVP_OBJECT_ADSPEC	*RSVP_ADSPEC*
RSVP_OBJECT_POLICY_INFO	*RSVP_POLICY_INFO*
QOS_OBJECT_END_OF_LIST	None. No more objects.

QOS priority

QOS priority defines the absolute priority of the flow. The priority levels themselves range from 0 to 7, lowest to highest. The *QOS_PRIORITY* structure is defined as

```
typedef struct _QOS_PRIORITY
{
    QOS_OBJECT_HDR  ObjectHdr;
    UCHAR           SendPriority
    UCHAR           SendFlags;
    UCHAR           ReceivePriority;
    UCHAR           Unused;
} QOS_PRIORITY, *LPQOS_PRIORITY;
```

These priority values determine the local priority (internal to the sending host computer) of the corresponding flow's traffic relative to traffic from other flows. The default priority for a flow is 3. This priority is used in combination with the *ServiceType* parameter of *FLOWSPEC* to determine the priority that should be applied to the flow internal to the Packet Scheduler. *SendFlags* and *ReceivePriority* are not currently used but might be used in the future.

QOS shape discard mode

This *QOS* object defines how the Packet Shaper element of Traffic Control processes the data of a given flow. This property most often comes into play when dealing with flows that do not conform to the parameters given in *FLOWSPEC*. That is, if an application is sending data at a rate faster than what is specified in the *TokenRate* field of the sending *FLOWSPEC*, it is considered nonconforming. This object defines how the local system handles this occurrence. The *QOS_SD_MODE* structure is defined as

```
typedef struct _QOS_SD_MODE
{
    QOS_OBJECT_HDR   ObjectHdr;
    ULONG            ShapeDiscardMode;
} QOS_SD_MODE, *LPQOS_SD_MODE;
```

The *ShapeDiscardMode* field can be one of the values specified in Table 12-4.

You might be wondering why you would want to use the *TC_NONCONF_DISCARD* mode when it might result in the dropping of data before it even gets sent on the wire. One such use is in sending audio or video data. In most cases, the *FLOWSPEC* structure is set up to reflect the sending of a packet whose size is equal to one frame of video or a small segment of audio. If for some reason the packet does not conform, is it better for an application to wait until it does conform (as is the case with *TC_NONCONF_SHAPE*), or should the application drop the packet altogether and move on to the next one? For time-critical data such as video, it is often better to drop the frame and move on.

Table 12-4. *QOS shape discard mode flags*

Flag	*Description*
TC_NONCONF_BORROW	The flow receives the resources remaining after all higher-priority flows have been serviced. Flows of this type are not subjected to either the Shaper or the Sequencer. If a value for *TokenRate* is specified, packets can be nonconforming and will be demoted to less than best-effort priority.
TC_NONCONF_SHAPE	A value for *TokenRate* must be specified. Nonconforming packets will be retained in the Packet Shaper until they become conforming.
TC_NONCONF_DISCARD	A value for *TokenRate* must be specified. Nonconforming packets will be discarded.

QOS traffic class

The *QOS_TRAFFIC_CLASS* structure can carry an 802.1p traffic class parameter provided to the host by a Layer 2 network device. The structure is defined as

```
typedef struct _QOS_TRAFFIC_CLASS
{
    QOS_OBJECT_HDR   ObjectHdr;
    ULONG            TrafficClass;
} QOS_TRAFFIC_CLASS, *LPQOS_TRAFFIC_CLASS;
```

Hosts stamp the MAC headers of corresponding transmitted packets with the *TrafficClass* value specified in the structure. Despite the structure's inclusion in Qos.h, applications are not allowed to set their own priority.

QOS destination address

The *QOS_DESTADDR* structure is used to specify the destination address for a connectionless sending socket without using a *WSAConnect* call. No RSVP PATH or RESV messages will be sent until the destination address of a connectionless socket is known. The destination address can be set with the *SIO_SET_QOS* ioctl command. The structure is defined as

```
typedef struct _QOS_DESTADDR
{
    QOS_OBJECT_HDR         ObjectHdr;
    const struct sockaddr *SocketAddress;
    ULONG                  SocketAddressLength;
} QOS_DESTADDR, *LPQOS_DESTADDR;
```

The *SocketAddress* field references the *SOCKADDR* structure that defines the endpoint's address for the given protocol. *SocketAddressLength* is simply the size of the *SOCKADDR* structure.

QOS shaper queue limit drop mode

This structure defines any overriding of the default schema used to drop packets when a flow's shaper queue limit is reached. Remember that the Traffic Shaper module of Traffic Control can be configured to discard any excess data pending if the currently queued data does not conform to the *FLOWSPEC* structure. The structure is defined as

```
typedef struct _QOS_SHAPER_QUEUE_LIMIT_DROP_MODE
{
    QOS_OBJECT_HDR   ObjectHdr;
    ULONG            DropMode;
}  QOS_SHAPER_QUEUE_LIMIT_DROP_MODE,
  *LPQOS_SHAPER_QUEUE_LIMIT_DROP_MODE;
```

The two possible values for *DropMode* are described in Table 12-5.

Table 12-5. *Shaper queue limit drop modes*

Flag	*Meaning*
QOS_SHAPER_DROP_FROM_HEAD	Drop packets from the head of the queue (the default behavior).
QOS_SHAPER_DROP_INCOMING	Drop any incoming packets once the queue limit is reached.

QOS shaper queue limit

The shaper queue limit structure allows the default per-flow limit on the shaper queue size to be overridden. The structure is defined as

```
typedef struct _QOS_SHAPER_QUEUE_LIMIT
{
    QOS_OBJECT_HDR   ObjectHdr;
    ULONG            QueueSizeLimit;
} QOS_SHAPER_QUEUE_LIMIT, *LPQOS_SHAPER_QUEUE_LIMIT;
```

QueueSizeLimit is the size of the shaper queue in bytes. Setting a larger shaper queue size can prevent data from being dropped by allowing data to be queued without running out of buffer space.

RSVP status info

The RSVP status info object is used to return RSVP-specific error and status information. The structure is defined as

```
typedef struct _RSVP_STATUS_INFO {
    QOS_OBJECT_HDR     ObjectHdr;
    ULONG              StatusCode;
    ULONG              ExtendedStatus1;
    ULONG              ExtendedStatus2;
} RSVP_STATUS_INFO, *LPRSVP_STATUS_INFO;
```

The *StatusCode* field is the RSVP message returned. The possible codes are described in Table 12-6. The other two fields, *ExtendedStatus1* and *ExtendedStatus2*, are reserved for provider-specific information.

Table 12-6. *RSVP status info codes*

Flag	*Meaning*
WSA_QOS_RECEIVERS	At least one RESV message has arrived.
WSA_QOS_SENDERS	At least one PATH message has arrived.
WSA_NO_QOS_RECEIVERS	There are no receivers.
WSA_NO_QOS_SENDERS	There are no senders.
WSA_QOS_REQUEST_CONFIRMED	The reserve has been confirmed.
WSA_QOS_ADMISSION_FAILURE	Request failed due to lack of resources.
WSA_QOS_POLICY_FAILURE	Request rejected for administrative reasons or bad credentials.
WSA_QOS_BAD_STYLE	Unknown or conflicting style.
WSA_QOS_BAD_OBJECT	There is a problem with some part of the *RSVP_FILTERSPEC* structure or the provider-specific buffer in general. (This object will be discussed shortly.)
WSA_QOS_TRAFFIC_CTRL_ERROR	There is a problem with some part of the *FLOWSPEC* structure.
WSA_QOS_GENERIC_ERROR	General error.
ERROR_IO_PENDING	Overlapped operation is canceled.

Typically, an application receives an *FD_QOS* event and calls *SIO_GET_QOS* to obtain a QOS structure containing an *RSVP_STATUS_INFO* object when an RSVP message is received. For example, for a QOS-enabled UDP-based receiver, an *FD_QOS* event containing a *WSA_QOS_SENDERS* message is generated to indicate that someone has requested the QOS service to send data to the receiver.

RSVP reserve info

The RSVP reserve info object is used for storing RSVP-specific information for fine-tuning interactions via the Winsock 2 QOS APIs and the provider-specific buffer. A *RSVP_RESERVE_INFO* object overrides the default reservation style and is used by a QOS receiver. The object is defined as

```
typedef struct _RSVP_RESERVE_INFO
{
    QOS_OBJECT_HDR    ObjectHdr;
    ULONG             Style;
    ULONG             ConfirmRequest;
    ULONG             NumPolicyElements;
    LPRSVP_POLICY     PolicyElementList;
    ULONG             NumFlowDesc;
    LPFLOWDESCRIPTOR  FlowDescList;
} RSVP_RESERVE_INFO, *LPRSVP_RESERVE_INFO;
```

The *Style* field specifies the filter type that should be applied to this receiver. Table 12-7 lists the filter types available and the default filter types used by different types of receivers. Each filter style will be discussed in greater detail shortly. If the *ConfirmRequest* field is nonzero, notification will be sent once the RESV request has been received for receiving applications. *NumPolicyElements* is related to the *PolicyElementList* field. It contains the number of *RSVP_POLICY* objects that are contained in the *PolicyElementList* field. (We define *RSVP_POLICY* a little bit later in this chapter.) Let's take a look at the different filter styles and the characteristics of each.

RSVP_DEFAULT_STYLE

This flag tells the QOS service provider to use the default style. Table 12-7 lists the default styles for the different possible receivers. Unicast receivers use fixed filter, whereas wildcard is for multicast receivers. UDP receivers that call *WSAConnect* also use fixed filter.

Table 12-7. *Default filter styles*

Filter Style	*Default Users*
Fixed filter	Unicast receivers Connected UDP receivers
Wildcard	Multicast receivers Unconnected UDP receivers
Shared explicit	None

RSVP_FIXED_FILTER_STYLE

Normally this style establishes a single flow with QOS guarantees between the receiver and a single source. This is the case for a unicast receiver and connected UDP receivers: *NumFlowDesc* is set to 1, and *FlowDescList* contains the sender's address. However, it is also possible to set up a multiple fixed filter style that allows a receiver to reserve mutually exclusive flows from multiple, explicitly identified sources. For example, if your receiver intends to receive data from three senders and needs guaranteed bandwidth of 20 Kbps for each, use the multiple fixed filter style. In this example, *NumFlowDesc* would be set to 3, while *FlowDescList* would contain three addresses, one for each *FLOWSPEC*. It is also possible to assign varying levels of QOS to each sender; they do not all have to be equal. Note that unicast receivers and connected UDP receivers cannot use multiple fixed filters. Figure 12-1 shows the relationship between *FLOWDESCRIPTOR* and *RSVP_FILTERSPEC* structures.

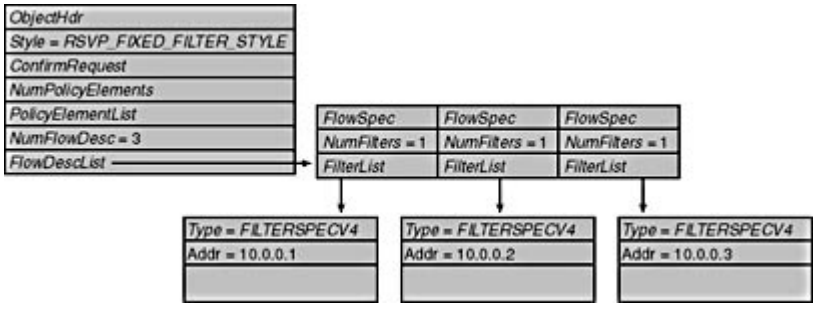

Figure 12-1. *Multiple fixed filter style*

RSVP_WILDCARD_STYLE

Multicast receivers and unconnected UDP receivers use the wildcard style. To use this style for TCP connections or for connected UDP receivers, set *NumFlowDesc* to 0 and *FlowDescList* to *NULL*. This is the default filter style for unconnected UDP receivers and multicast applications because the sender's address is unknown.

RSVP_SHARED_EXPLICIT_STYLE

This style is somewhat similar to multiple fixed filter style except that network resources, instead of being allocated for each sender, are shared among all senders. In this case, *NumFlowDesc* is 1 and *FlowDescList* contains the list of sender addresses. Figure 12-2 illustrates this style.

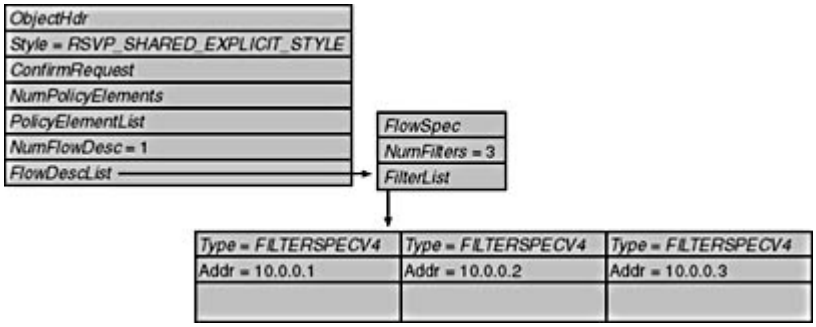

Figure 12-2. *Shared explicit style*

We've introduced the last two fields, *NumFlowDesc* and *FlowDescList*, in our discussion of RSVP styles. How you use these two fields depends on the style. *NumFlowDesc* defines the number of *FLOWDESCRIPTOR* objects in the *FlowDescList* field. This structure is defined as

```
typedef struct _FLOWDESCRIPTOR
{
    FLOWSPEC             FlowSpec;
    ULONG                NumFilters;
    LPRSVP_FILTERSPEC    FilterList;
} FLOWDESCRIPTOR, *LPFLOWDESCRIPTOR;
```

This object is used to define the types of filters per *FLOWSPEC* given by *FlowSpec*. Again, the *NumFilters* field contains the number of *RSVP_FILTERSPEC* objects present in the *FilterList* array. The *RSVP_FILTERSPEC* object is defined as

```
typedef struct _RSVP_FILTERSPEC {
    FilterType   Type;
    union {
        RSVP_FILTERSPEC_V4      FilterSpecV4;
        RSVP_FILTERSPEC_V6      FilterSpecV6;
        RSVP_FILTERSPEC_V6_FLOW FilterSpecV6Flow;
        RSVP_FILTERSPEC_V4_GPI  FilterSpecV4Gpi;
        RSVP_FILTERSPEC_V6_GPI  FilterSpecV6Gpi;
    };
} RSVP_FILTERSPEC, *LPRSVP_FILTERSPEC;
```

The first field, *Type*, is a simple enumeration of the following values:

```
typedef enum {
        FILTERSPECV4 = 1,
        FILTERSPECV6,
        FILTERSPECV6_FLOW,
        FILTERSPECV4_GPI,
        FILTERSPECV6_GPI,
        FILTERSPEC_END
} FilterType;
```

This enumeration specifies the object present in the union. Each of these filter specs is defined below.

```
typedef struct _RSVP_FILTERSPEC_V4 {
    IN_ADDR_IPV4    Address;
    USHORT          Unused;
    USHORT          Port;
} RSVP_FILTERSPEC_V4, *LPRSVP_FILTERSPEC_V4;

typedef struct _RSVP_FILTERSPEC_V6 {
    IN_ADDR_IPV6    Address;
    USHORT          UnUsed;
    USHORT          Port;
} RSVP_FILTERSPEC_V6, *LPRSVP_FILTERSPEC_V6;

typedef struct _RSVP_FILTERSPEC_V6_FLOW {
    IN_ADDR_IPV6    Address;
    UCHAR           UnUsed;
    UCHAR           FlowLabel[3];
} RSVP_FILTERSPEC_V6_FLOW, *LPRSVP_FILTERSPEC_V6_FLOW;

typedef struct _RSVP_FILTERSPEC_V4_GPI {
    IN_ADDR_IPV4    Address;
    ULONG           GeneralPortId;
} RSVP_FILTERSPEC_V4_GPI, *LPRSVP_FILTERSPEC_V4_GPI;

typedef struct _RSVP_FILTERSPEC_V6_GPI {
    IN_ADDR_IPV6    Address;
    ULONG           GeneralPortId;
} RSVP_FILTERSPEC_V6_GPI, *LPRSVP_FILTERSPEC_V6_GPI;
```

RSVP Adspec

The *RSVP_ADSPEC* object defines the information carried in the RSVP Adspec. This RSVP object typically indicates which service types are available (controlled load or guaranteed), whether a non-RSVP hop has been encountered by the PATH message, and the minimum MTU along the path. The structure is defined as

```
typedef struct _RSVP_ADSPEC
{
    QOS_OBJECT_HDR     ObjectHdr;
    AD_GENERAL_PARAMS  GeneralParams;
    ULONG              NumberOfServices;
    CONTROL_SERVICE    Services[1];
} RSVP_ADSPEC, *LPRSVP_ADSPEC;
```

The first field of interest is *GeneralParams*, which is a structure of type *AD_GENERAL_PARAMS*. This structure is exactly as it sounds—it defines some general characterization parameters. The definition of this object is

```
typedef struct _AD_GENERAL_PARAMS
{
    ULONG       IntServAwareHopCount;
    ULONG       PathBandwidthEstimate;
    ULONG       MinimumLatency;
    ULONG       PathMTU;
    ULONG       Flags;
} AD_GENERAL_PARAMS, *LPAD_GENERAL_PARAMS;
```

The *IntServAwareHopCount* is the number of hops that conform to Integrated Services (IntServ) requirements. *PathBandwidthEstimate* is the minimum bandwidth available from sender to receiver. *MinimumLatency* is the sum of minimum latencies, in microseconds, of the packet forwarding processes in the routers. *PathMTU* is the maximum transmission unit—end-to-end—that will not incur any fragmentation. The *Flags* field is not currently used.

RSVP policy info

The last provider object we'll take a look at is the RSVP policy info. This object is rather nebulous—it contains any number of policy elements from RSVP that are not defined. The structure is defined as

```
typedef struct _RSVP_POLICY_INFO  {
    QOS_OBJECT_HDR     ObjectHdr;
    ULONG              NumPolicyElement;
    RSVP_POLICY        PolicyElement[1];
} RSVP_POLICY_INFO, *LPRSVP_POLICY_INFO;
```

The *NumPolicyElement* field gives the number of *RSVP_POLICY* structures present in the *PolicyElement* array. This structure is defined as

```
typedef struct _RSVP_POLICY {
    USHORT  Len;
    USHORT  Type;
    UCHAR   Info[4];
} RSVP_POLICY, *LPRSVP_POLICY;
```

The *RSVP_POLICY* structure is data transported by RSVP on behalf of the policy component and is not particularly relevant to our needs.

Programming QOS

Central to QOS is the initiation of an RSVP session. It's not until the RSVP PATH and RESV messages have been sent and processed that bandwidth is reserved for the process. Knowing when RSVP messages are generated is important to applications. For senders, three parameters must be known before a PATH message is generated:

- Sending *FLOWSPEC* member
- Source IP address and port
- Destination IP address, port, and protocol

The *FLOWSPEC* member is known whenever a QOS-enabled function is called, such as *WSAConnect*, *WSAJoinLeaf*, *WSAIoctl* (with the *SIO_SET_QOS* option), and so on. The source IP address and port will not be known until the socket is bound locally, either implicitly (such as by connecting) or explicitly by bind. Finally, the application needs the data's destination. This information is gathered upon either a connect call or, in the case of connectionless UDP, the setting of the *QOS_DESTADDR* object in the provider-specific data passed using the *SIO_SET_QOS* ioctl command.

Similarly, for an RSVP RESV message to be generated, three things must be known:

- Receiving *FLOWSPEC* member
- Address and port of each sender
- Local address and port of the receiving socket

The receiving *FLOWSPEC* member is obtained from any of the QOS-enabled Winsock functions. The address and port of each sender depend on the filter style, which can be set manually via the *RSVP_RESERVE_INFO* provider-specific structure, discussed earlier. Otherwise, this information is obtained from a PATH message. Of course, depending on the socket type, it is not always necessary to have already received a PATH statement to obtain the sender's address to generate RESV messages. The wildcard filter style used in multicasting is an example of this. The RESV message sent applies to all senders in the session. The local address and port are self-explanatory for unicast and UDP receivers but not for multicast receivers. In the case of multicast receivers, the local address and port are the multicast address and its corresponding port number.

In this section, we'll first cover the different socket types and their interaction with the QOS service provider and RSVP messages. Then we'll take a look at how the QOS service provider notifies applications of certain events. Understanding these concepts is central to writing successful QOS-enabled applications. Programming such applications is a matter of knowing how to obtain QOS guarantees as well as knowing when those guarantees are put into effect and when and how they can change.

RSVP and Socket Types

We now have a basic understanding of how PATH and RESV RSVP messages are generated. In the following sections, we'll look at the different types of sockets—UDP, TCP, and multicast UDP—and how they interact with the QOS service provider to generate PATH and RESV messages.

Unicast UDP

Because you have the option of using either connected or unconnected UDP sockets, setting QOS on unicast UDP sockets presents quite a few options. In the case of the UDP sender, the sending *FLOWSPEC* is obtained from one of the QOS-invoking functions. The local address and port are obtained either from an explicit bind call or from an implicit bind done by *WSAConnect*. The last piece is the address and port of the receiving application, which can be specified either in *WSAConnect* or via the *QOS_DESTADDR* provider-specific structure passed through the *SIO_SET_QOS* option. Be aware that if *SIO_SET_QOS* is used to set QOS, the socket must be bound beforehand.

For the UDP receiver, *WSAConnect* can be called to limit the receiving application to a single sender. Additionally, applications can specify a *QOS_DESTADDR* structure with the *SIO_SET_QOS* ioctl command. Otherwise, the

SIO_SET_QOS can be called without providing any kind of destination address. In this case, a RESV message will be generated with the wildcard filter style. In fact, specifying the destination address via *WSAConnect* or via the *QOS_DESTADDR* structure should be done only if you want the application to receive data from only one sender that uses the fixed filter style.

The UDP receiver can actually call both *WSAConnect* and the *SIO_SET_QOS* ioctl command in any order. If *SIO_SET_QOS* is called before *WSAConnect*, a RESV message is created with the wildcard filter first. Once the connect call is made, the previous RESV session is torn down and a new one is generated with the fixed filter style. Alternatively, calling *SIO_SET_QOS* after *WSAConnect* and a fixed filter RESV message does not negate the RSVP session and generate a wildcard filter style. Instead it simply updates the QOS parameters associated with the existing RSVP session.

Unicast TCP

TCP sessions have two possibilities. First, the sender can be the client who connects to the server and sends data. The second possibility is that the server that the client connects to might be the sender. In the case of the client, QOS parameters can be specified directly in the *WSAConnect* call, which will result in PATH messages being sent. The ioctl command *SIO_SET_QOS* can also be called before calling connect, but until one of the connect calls knows the destination address, no PATH messages will be generated.

In the case in which the sender is the server, the server calls *WSAAccept* to accept the client connection. This function does not provide a means of setting QOS on the accepted socket. If QOS is set before a call to *WSAAccept* by using *SIO_SET_QOS*, any accepted socket inherits the QOS levels set on the listening socket. Note that if the sender uses the conditional function in *WSAAccept*, the function should pass QOS values set on the connecting client. However, this is not the case. The QOS service provider passes junk, which is the behavior on both Windows 98 and Windows 2000. The exception is that if the *lpSQOS* parameter is non-null under Windows 98, some kind of QOS values must be set via the *SIO_SET_QOS* ioctl command within the conditional function; otherwise, the *WSAAccept* call fails even if *CF_ACCEPT* is returned. QOS can also be set on the client socket after it has been accepted.

Let's look at receiving TCP applications. The first case is calling *WSAConnect* with a receiving *FLOWSPEC*. When this occurs, the QOS service provider creates a RESV request. If QOS parameters are not supplied to *WSAConnect*, the *SIO_SET_QOS* ioctl command can be set at a later time (resulting in a RESV message). The last combination is the server being the receiver, which is similar to the sending case. QOS can be set on the listening socket before a *WSAAccept* call, in which case the client socket inherits the same QOS levels. Otherwise, QOS can be set in the conditional function or after the socket has been accepted. In either case, the QOS service provider generates a RESV message as soon as a PATH message arrives.

Multicast

Multicast senders behave the same way as UDP senders except that *WSAJoinLeaf* is used to become a member of the multicast group, as opposed to calling *WSAConnect* with the destination address. QOS can be set with *WSAJoinLeaf* or separately through an *SIO_SET_QOS* call. The multicast session address is used to compose the RSVP session object included in the RSVP PATH message.

In the case of the multicast receiver, no RESV messages will be generated until the multicast address is specified via the *WSAJoinLeaf* function. Because the multicast receiver doesn't specify a peer address, the QOS provider generates RESV messages with the wildcard filter style. The QOS service provider does not prohibit a socket from joining multiple multicast groups. In this case, the service provider sends RESV messages for all groups that have a matching PATH message. The QOS parameters supplied to each *WSAJoinLeaf* will be used in each RESV message, but if *SIO_SET_QOS* is called on the socket after joining multiple groups, the new QOS parameters will be applied to all multicast groups joined.

When a sender sends data to a multicast group, only data sent to the multicast group that the sender joined results in QOS being applied to that data. In other words, if you join one multicast group and use *sendto/WSASendTo* with any other multicast group as the destination, QOS is not applied to that data. Additionally, if a socket joins a multicast group specifying a particular direction (for example, using *JL_SENDER_ONLY* or *JL_RECEIVER_ONLY* in the *dwFlags* parameter to *WSAJoinLeaf*), QOS is applied accordingly. A socket set as a receiver only will not gain any QOS benefits for sent data.

QOS Notifications

Thus far, you have learned how to invoke QOS for TCP, UDP, and multicast UDP sockets and the corresponding RSVP events that occur depending on whether you're sending or receiving. However, the completion of these RSVP messages is not strictly tied to the API calls that invoke them. That is, issuing a *WSAConnect* call for a TCP receiving socket generates a RESV message, but the RESV message is independent of the actual API call in that the call returns without any assurances that the reservation is approved and network resources are allocated. Because of this, a new asynchronous event has been added, *FD_QOS*, which is posted to a socket. Typically, an *FD_QOS* event notification will be posted in the following events.

- Notification of the acceptance or rejection of the application's QOS request
- Significant changes in the QOS provided by the network (as opposed to previously negotiated values)
- Status regarding whether a QOS peer is ready to send or receive data for a particular flow

Registering for *FD_QOS* notifications

To take advantage of these notifications, an application must register to be notified when an *FD_QOS* event occurs. There are a couple ways to do this. First you can use either *WSAEventSelect* or *WSAAsyncSelect* and include the *FD_QOS* flag in the bitwise ORing of event flags. However, an application is eligible to receive the *FD_QOS* event only if a call has already been made to one of the QOS invoking functions. Note that in some cases an application might want to receive the *FD_QOS* event without having to set QOS levels on a socket. This can be accomplished by setting up a *QOS* structure whose sending and receiving *FLOWSPEC* members contain either the *QOS_NOT_SPECIFIED* or the *SERVICETYPE_NOTRAFFIC* flag. The only catch is that the *SERVICE_NO_QOS_SIGNALING* flag must be ORed with the *SERVICETYPE_NOTRAFFIC* flag for the direction of QOS in which you want to receive event notification.

If you need exact information on how to call the two asynchronous select functions, consult Chapter 8, which covers them in great detail. If you use *WSAEventSelect* once the event has been triggered, call the *WSAEnumNetworkEvents* function to obtain additional status codes that might be available. This function is also covered in Chapter 8, but we'll review it here since it's short, simple, and important to QOS applications. Pass the socket handle, the event handle, and a *WSANETWORKEVENTS* object into the call, which will return and set event information into the supplied structure. This structure is defined as

```
typedef struct _WSANETWORKEVENTS
{
    long   lNetworkEvents;
    int    iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

The *lNetworkEvents* field will be set to a bitwise ORing of all the event flags triggered. To detect the occurrence of a particular event, simply AND (with the & operator) this field with the event flag. If the result is nonzero, that event was triggered. The *iErrorCode* array is used to indicate errors or, in the case of QOS, status information. If an event is triggered, a flag corresponding to that event is used to index into this array. If the value of the array index is 0, no error occurred; otherwise, the value is an index to the array element that contains the error code. For example, if the *FD_QOS* event was triggered, use the *FD_QOS_BIT* flag to index into the *iErrorCode* array to check for any errors or status information. All other Winsock asynchronous events (*FD_READ_BIT*, *FD_WRITE_BIT*, and so on) have similarly defined index flags.

RSVP notifications

We mentioned earlier that there are a couple of ways to receive QOS notifications. This information actually ties into this section: obtaining the results of a QOS event. If you have registered to receive *FD_QOS* notifications with either *WSAAsyncSelect* or *WSAEventSelect* and you actually receive an *FD_QOS* event notification, you must perform a call to *WSAIoctl* with the *SIO_GET_QOS* ioctl option to find out what triggered the event. You don't actually have to register for *FD_QOS* events—you can simply call *WSAIoctl* with the *SIO_GET_QOS* command using overlapped I/O. This also requires that you specify a completion routine, which is invoked once the QOS service provider detects a change in QOS. Once the callback occurs, a QOS structure will be available in the output buffer.

In either case, once a change in QOS has occurred, your application can be notified of this change by registering for *FD_QOS* or by using overlapped I/O and *SIO_GET_QOS*. If you register for *FD_QOS*, you also want to call *WSAIoctl* with the *SIO_GET_QOS* ioctl command upon event notification. For both methods, the QOS structure returned contains QOS information for only a single direction. That is, the *FLOWSPEC* structure for the invalid direction has its *ServiceType* field set to *SERVICETYPE_NOCHANGE*. Additionally, more than one QOS event might have occurred, in which case you should call *WSAIoctl* and *SIO_GET_QOS* in a loop until *SOCKET_ERROR* is returned and *WSAGetLastError* returns *WSAEWOULDBLOCK*. The final concern when calling *SIO_GET_QOS* is the buffer size. When an *FD_QOS* event has been triggered, it is possible that provider-specific objects will be returned. In fact, the *RSVP_STATUS_INFO* structure will most often be returned, provided the buffer is large enough. See the earlier entry on *WSAIoctl* for information on how to find the right-size buffer.

If your application uses one of the asynchronous event functions, a particularly important issue is that once an *FD_QOS* event occurs, you must *always* perform an *SIO_GET_QOS* operation to reenable *FD_QOS* notifications.

You now know how to receive QOS event notifications and obtain new QOS parameters as a result of these events, but what types of notifications will occur? The first and most obvious reason for a QOS event is a change in the *FLOWSPEC* parameters for a given flow. For example, if you set up a socket with best effort service, periodically the QOS service provider will send notification to your application indicating the current conditions on the network. Additionally, if you specify controlled load as well as other parameters, the QOS parameters for token bucket size and token rate might change slightly from what you requested once the reservation occurs. Your application should compare the *FLOWSPEC* returned once a QOS notification occurs to what you originally requested to ensure that it is sufficient for your application to continue. Also remember that throughout the life of a QOS-enabled socket, you can always perform a *SIO_SET_QOS* to change any of the parameters, which will result in a QOS notification for the peer or peers associated with your current RSVP session. A robust application should be able to handle these conditions.

In addition to updating QOS parameters, QOS event notification signals other occurrences, such as notification of senders or receivers. The possible events are listed in Table 12-6. There are two ways to obtain these status codes. The first is as a part of the *RSVP_STATUS_INFO* object. When a QOS event occurs and a call is made to *SIO_GET_QOS*, it is possible that a *RSVP_STATUS_INFO* object will be returned as part of the provider-specific buffer. Second, if you use *WSAEventSelect* to register for events, these codes can be returned in the *WSANETWORKEVENTS* structure returned from *WSAEnumNetworkEvents*. The codes defined in Table 12-6 can be found in the *iErrorCode* array, indexed by *FD_QOS_BIT*. The first five codes listed are not error codes. They return valuable information concerning the status of the QOS connection. The other status codes listed in the table are QOS errors of concern, but they won't prevent you from sending and receiving data—they merely indicate an error in the QOS session. Of course, data sent in this situation will not carry any of the requested QOS guarantees.

WSA_QOS_RECEIVERS and *WSA_QOS_NO_RECEIVERS*

In the case of unicast, after a sender starts up and receives the first RESV message, a *WSA_QOS_RECEIVERS* is passed up to the application. If the receiver performs any of the steps to disable QOS, the result is a RESV teardown message. Once the sender receives this, *WSA_QOS_NO_RECEIVERS* is passed up to the application. Of course, with unicast many receivers simply close the socket, generating both an *FD_CLOSE* event and the *WSA_QOS_NO_RECEIVERS* event. In most cases, an application's response is simply to close the sending socket.

In the case of multicast, the sending application receives *WSA_QOS_RECEIVERS* whenever the number of receivers changes and is nonzero. In other words, a single multicast sender receives *WSA_QOS_RECEIVERS* every time a QOS receiver joins the group, as well as every time a receiver drops out of a group—as long as at least one receiver remains.

WSA_QOS_SENDERS and *WSA_QOS_NO_SENDERS*

The senders notification is similar to the receivers event except that it deals with the receipt of the PATH message. For unicast receivers after startup, the receipt of the first PATH message generates *WSA_QOS_SENDERS*, while the PATH teardown message initiates a *WSA_QOS_NO_SENDERS* message.

Likewise, multicast receivers receive the *WSA_QOS_SENDERS* notification whenever the number of senders decrements or increments and is nonzero. Once the number of senders reaches 0, the *WSA_QOS_NO_SENDERS* message is passed to the application.

WSA_QOS_REQUEST_CONFIRMED

This last status message is issued to receiving QOS applications if they ask to be notified when a reservation request has been confirmed. Within the *RSVP_STATUS_INFO* structure is a field, *ConfirmRequest*, which if set to nonzero informs the QOS service provider to notify the application when the reservation request has been confirmed. This object is a provider-specific option that can be passed along with a QOS structure to the *SIO_SET_QOS* ioctl command.

QOS Templates

Winsock provides several predefined QOS structures, referred to as templates, that an application can query by name. These templates define the QOS parameters for some common audio and video codecs, such as G711 and H263QCIF. The function *WSAGetQOSByName* is defined as

```
BOOL WSAGetQOSByName(
    SOCKET s,
    LPWSABUF lpQOSName,
    LPQOS lpQOS
);
```

If you don't know the name of the installed templates, you can use this function to first enumerate all template names. This is accomplished by providing a sufficiently large buffer in *lpQOSName* with its first character set to the null character and passing a null pointer for *lpQOS*, as in the following code:

```
WSABUF  wbuf;
char    cbuf[1024];

cbuf[0] = '\0';
wbuf.buf = cbuf;
wbuf.len = 1024;
WSAGetQOSByName(s, &wbuf, NULL);
```

Upon return, the character buffer is filled with an array of strings separated by a null character, and the entire list is terminated by another null character. This means the last string entry will have two consecutive null characters. From here you can get the names of all the installed templates and query for a specific one. The following code looks up the G711 template:

```
QOS     qos;
WSABUF  wbuf;

wbuf.buf = "G711";
wbuf.len = 4;
WSAGetQOSByName(s, &wbuf, &qos);
```

In the event that the requested QOS template does not exist, the lookup returns *FALSE* and the error is *WSAEINVAL*. Upon success, the function returns *TRUE*. The example *Qostemplate.c* on the accompanying CD illustrates how to enumerate the installed QOS templates.

Additionally, you can install your own QOS template so that other applications can query for it by name. Two functions—*WSCInstallQOSTemplate* and *WSCRemoteQOSTemplate*—do this. The first function installs a QOS template, and the second removes it. The prototypes are

```
BOOL WSCInstallQOSTemplate(
    const LPGUID lpProviderId,
    LPWSABUF lpQOSName,
    LPQOS lpQOS
);

BOOL WSCRemoveQOSTemplate(
    const LPGUID lpProviderId,
    LPWSABUF lpQOSName
);
```

These two functions are fairly self-explanatory. To install a template, call *WSCInstallQOSTemplate* with a GUID, the name of the template, and the QOS parameters. The GUID is a unique identifier for this template that can be generated by such utilities as Uuidgen.exe. To remove the template, simply supply the template name—along with the same GUID used in the installation process—to *WSCRemoveQOSTemplate*. Both functions return *TRUE* when successful.

Examples

In this section, we'll take a look at three programming examples of how to use QOS. The first example, which uses TCP, is the most straightforward because it is connection-oriented. The second example uses UDP and does not use any connect calls. The last example uses multicast UCP. In all three examples, we will use *WSAEventSelect* because it's a bit simpler than *WSAAsyncSelect* . While we include the entire TCP unicast example, we include only important code snippets for the UDP and multicasting examples because many of the concepts are the same, no matter what type of socket is used. See the accompanying CD for the entire example. All three examples rely on a couple of support routines, *PrintQos* and *FindProtocolInfo* , which are defined in the files Printqos.c and Provider.c, respectively. The former routine simply prints out the contents of a QOS structure, while the latter finds a protocol from the provider catalog with the required attributes, such as QOS.

Unicast TCP

The example for unicast TCP QOS is given in Figure 12-3. The code for this example can be found in the Chapter 12 folder as Qostcp.c. The example is a bit long, but not particularly complicated. Most of the code is nothing more than the usual *WSAEventSelect* code that we introduced in Chapter 8 . The only exception is what we do in the case of an *FD_QOS* event. The main function doesn't do anything out of the ordinary. The arguments are parsed, a socket is created, and either the *Server* or the *Client* function is called—depending on whether the application is called as the server or the client. Let's examine the client connection first.

In all three examples, a command line parameter tells the example when to set QOS: before connection, during connection, after connection, or after the peer requests QOS to set QOS locally. (The command line parameters for Qostp.c are listed in Table 12-8.) If QOS is selected to be set before connection (for the client), bind the socket to a random port and then call *SIO_SET_QOS* with a sending QOS *FLOWSPEC* . Note that it isn't really necessary to bind before calling *SIO_SET_QOS* because the peer's address is not known until a connect call is made, and an RSVP session cannot be initiated until the peer's address is known.

If the user elects to set QOS during connection, the example code passes the QOS structure into the *WSAConnect* call. This call initiates an RSVP session and connects the client to the specified server. Otherwise, the user specifies that the example should wait for the peer to set QOS, and no QOS structure is passed to *WSAConnect* . Instead the code takes the sending QOS structure, ORs in the *SERVICE_NO_QOS_SIGNALING* flag to the *ServiceType* field in the *FLOWSPEC* structures, and calls *WSAIoctl* with the *SIO_SET_FLAG* ioctl command. This tells the QOS service provider not to invoke Traffic Control but to still look for RSVP messages.

After QOS is set, the events that the client wants to be notified of are registered, including *FD_QOS* . Notice that QOS must be set on the socket beforehand in order for the application to request receiving *FD_QOS* . Once this occurs, the client waits in a loop on *WSAWaitForMultipleEvents* , which unblocks when one of the selected events is signaled. Once an event occurs, the events are enumerated along with any errors in *WSAEnumNetworkEvents* .

For the most part, Qostcp.c handles the other events, such as *FD_READ* , *FD_WRITE* , and *FD_CLOSE* , in the same way as the *WSAEventSelect* example code in Chapter 8 . The only item of note is in the *FD_WRITE* event. One of the command line options is to wait until an RSVP PATH message has been received before sending the data. This is especially relevant if the data being transmitted is likely to exceed the best-effort bandwidth available on the network. The *AbleToSend* function calls *SIO_CHK_QOS* to determine whether the QOS parameters requested are within the available best-effort limits. If so, it should be OK to start sending data; otherwise, wait for a confirmation to send data.

In our client's case, we want to receive the *WSA_QOS_RECEIVERS* message to indicate the receipt of a RESV message. This can be indicated upon receipt of an *FD_QOS* event. At this point, we call the *SIO_CHK_QOS* command to obtain status information. This *WSA_QOS_RECEIVERS* flag can be returned in two ways. First the flag can be returned in the *iErrorCode* field of the *WSANETWORKEVENTS* structure as the element indexed by *FD_QOS_BIT* . Second an *RSVP_STATUS_INFO* structure can be returned in the buffer passed to *WSAIoctl* using the *SIO_GET_QOS* ioctl command. This structure too might contain the *WSA_QOS_RECEIVERS* flag in its *StatusCode* field. If the wait to send flag has been specified, we check the error field from *WSANETWORKEVENTS* to see whether an *RSVP_STATUS_INFO* structure has been returned. If the flag is present, we send data. That's all! The code necessary to support QOS for the client is straightforward.

The server side of the example is a bit more complicated, but only because it needs to manage zero or more client connections. The listening socket and the client sockets are handled in a single array, *sc* . Array element 0 is the listening socket, while the rest are possible client connections. The global variable *nConns* contains the number of current clients. Whenever a client connection finishes, all active sockets are compacted toward the beginning of the socket array. There is also a corresponding array of event handles.

The server first binds the listening socket and sets receiving QOS if the user chooses to set QOS before accepting

client connections. Any QOS parameters set on the listening socket are copied to the client connection (unless the server is using *AcceptEx*). The listening socket registers to receive only *FD_ACCEPT* events. The rest of the server routine is a loop that waits for events on the array of socket handles. At first the only socket in the array is the listening socket, but as more client connections are established there will be more sockets and their corresponding events. If *WSAWaitForMultipleEvents* unblocks as a result of an event and indicates the event handle in array element 0, the event is occurring on the listening socket. If so, the code calls *WSAEnumNetworkEvents* to find out which event is occurring. If the event is occurring on a client socket, the code calls the handler routine *HandleClientEvents* .

On the listening socket, the event of interest is *FD_ACCEPT* . When this event happens, *WSAAccept* is called with a conditional function. Remember that the QOS parameters passed into the conditional function can't be trusted, and if the QOS parameter is non-null on Windows 98, some sort of QOS must be set. Windows 2000 does not have that limitation; QOS can be set at any time. If the user specifies that QOS be set during the accept call, this occurs within the conditional function. Once the client socket is accepted, a corresponding event handle is created and the appropriate events are registered for the socket.

The function *HandleClientEvents* handles any events occurring on the client sockets. The read and write events are straightforward—the only exception is whether to wait for the reservation confirmation before sending. If the user specifies to wait for the reservation confirmation to arrive before sending data, the client waits for the *WSA_QOS_RECEIVERS* message to be returned in an *FD_QOS* event. If the message returns, the sending of the data doesn't occur until *FD_QOS* is received. Usually the most significant aspect of this example is the setting of QOS on the socket and the *FD_QOS* handler.

Figure 12-3. *Unicast TCP example (Qostcp.c)*

```
// Module: Qostcp.c
//
#include <winsock2.h>
#include <windows.h>
#include <qos.h>
#include <qossp.h>

#include "provider.h"
#include "printqos.h"

#include <stdio.h>
#include <stdlib.h>

#define QOS_BUFFER_SZ       16000 // Default buffer size for
                                  // SIO_GET_QOS
#define DATA_BUFFER_SZ       2048 // Send/Recv buffer size

#define SET_QOS_NONE            0   // No QOS
#define SET_QOS_BEFORE          1   // Set QOS on listening socket
#define SET_QOS_DURING          2   // Set QOS in conditional accept
#define SET_QOS_AFTER           3   // Set QOS after accept
#define SET_QOS_EVENT           4   // Wait for FD_QOS and then set

#define MAX_CONN                10

int  iSetQos,          // When to set QOS?
     nConns;
BOOL bServer,          // Client or server?
     bWaitToSend,      // Wait to send data until RESV
     bConfirmResv;
char szServerAddr[64]; // Server's address
QOS  clientQos,        // QOS client structure
```

```
     serverQos;        // QOS server structure
RSVP_RESERVE_INFO  qosreserve;

//
// Set up some common FLOWSPEC structures
//
const FLOWSPEC flowspec_notraffic = {QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED,
                                     SERVICETYPE_NOTRAFFIC,
                                     QOS_NOT_SPECIFIED,
                                     QOS_NOT_SPECIFIED};

const FLOWSPEC flowspec_g711 = {8500,
                                680,
                                17000,
                                QOS_NOT_SPECIFIED,
                                QOS_NOT_SPECIFIED,
                                SERVICETYPE_CONTROLLEDLOAD,
                                340,
                                340};

const FLOWSPEC flowspec_guaranteed = {17000,
                                      1260,
                                      34000,
                                      QOS_NOT_SPECIFIED,
                                      QOS_NOT_SPECIFIED,
                                      SERVICETYPE_GUARANTEED,
                                      340,
                                      340};

//
// Function: SetReserveInfo
//
// Description:
//    For receivers, if a confirmation is requested this must be
//    done with an RSVP_RESERVE_INFO structure
//
void SetQosReserveInfo(QOS *lpqos)
{
    qosreserve.ObjectHdr.ObjectType = RSVP_OBJECT_RESERVE_INFO;
    qosreserve.ObjectHdr.ObjectLength = sizeof(RSVP_RESERVE_INFO);
    qosreserve.Style = RSVP_DEFAULT_STYLE;
    qosreserve.ConfirmRequest = bConfirmResv;
    qosreserve.NumPolicyElement  = 0;
    qosreserve.PolicyElementList = NULL;
    qosreserve.FlowDescList = NULL;

    lpqos->ProviderSpecific.buf = (char *)&qosreserve;
    lpqos->ProviderSpecific.len = sizeof(qosreserve);
```

```
    return;
}

//
// Function: InitQos
//
// Description:
//    Set up the client and server QOS structures. This is
//    broken out into a separate function so that you can change
//    the requested QOS parameters to see how that affects
//    the application.
//
void InitQos()
{
    clientQos.SendingFlowspec = flowspec_g711;
    clientQos.ReceivingFlowspec =  flowspec_notraffic;
    clientQos.ProviderSpecific.buf = NULL;
    clientQos.ProviderSpecific.len = 0;

    serverQos.SendingFlowspec = flowspec_notraffic;
    serverQos.ReceivingFlowspec = flowspec_g711;
    serverQos.ProviderSpecific.buf = NULL;
    serverQos.ProviderSpecific.len = 0;
    if (bConfirmResv)
        SetQosReserveInfo(&serverQos);
}

//
// Function: usage
//
// Description:
//    Print out usage information
//
void usage(char *progname)
{
    printf("usage: %s -q:x -s -c:IP\n", progname);
    printf("      -q:[b,d,a,e] When to request QOS\n");
    printf("          b        Set QOS before bind or connect\n");
    printf("          d        Set QOS during accept cond func\n");
    printf("          a        Set QOS after session setup\n");
    printf("          e        Set QOS only upon receipt of FD_QOS\n");
    printf("      -s           Act as server\n");
    printf("      -c:Server-IP Act as client\n");
    printf("      -w           Wait to send until RESV has arrived\n");
    printf("      -r           Confirm reservation request\n");
    ExitProcess(-1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse command line arguments and set global variables to
```

```
//    indicate how the application should act
//
void ValidateArgs(int argc, char **argv)
{
    int      i;

    // Initialize globals to a default value
    //
    iSetQos = SET_QOS_NONE;
    bServer = TRUE;
    bWaitToSend = FALSE;
    bConfirmResv = FALSE;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'q':        // When to set QOS
                    if (tolower(argv[i][3]) == 'b')
                        iSetQos = SET_QOS_BEFORE;
                    else if (tolower(argv[i][3]) == 'd')
                        iSetQos = SET_QOS_DURING;
                    else if (tolower(argv[i][3]) == 'a')
                        iSetQos = SET_QOS_AFTER;
                    else if (tolower(argv[i][3]) == 'e')
                        iSetQos = SET_QOS_EVENT;
                    else
                        usage(argv[0]);
                    break;
                case 's':        // Server
                    printf("Server flag set!\n");
                    bServer = TRUE;
                    break;
                case 'c':        // Client
                    printf("Client flag set!\n");
                    bServer = FALSE;
                    if (strlen(argv[i]) > 3)
                        strcpy(szServerAddr, &argv[i][3]);
                    else
                        usage(argv[0]);
                    break;
                case 'w':       // Wait to send data until
                                // RESV has arrived
                    bWaitToSend = TRUE;
                    break;
                case 'r':
                    bConfirmResv = TRUE;
                    break;
                default:
                    usage(argv[0]);
                    break;
```

```
            }
        }
    }
    return;
}

//
// Function: AbleToSend
//
// Description:
//    Checks to see whether data can be sent on the socket before
//    any RESV messages have arrived. This function checks to see whether
//    the best-effort level currently available on the network is
//    sufficient for the QOS levels that were set on the socket.
//
BOOL AbleToSend(SOCKET s)
{
    int     ret;
    DWORD   dwCode = ALLOWED_TO_SEND_DATA,
            dwValue,
            dwBytes;

    ret = WSAIoctl(s, SIO_CHK_QOS, &dwCode, sizeof(dwCode),
                &dwValue, sizeof(dwValue), &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl() failed: %d\n", WSAGetLastError());
        return FALSE;
    }
    return (BOOL)dwValue;
}

//
// Function: ChkForQosStatus
//
// Description:
//    Check for the presence of an RSVP_STATUS_INFO object and
//    determine whether the supplied flags are present in that
//    object
//

DWORD ChkForQosStatus(QOS *lpqos, DWORD dwFlags)
{
    QOS_OBJECT_HDR   *objhdr = NULL;
    RSVP_STATUS_INFO *status = NULL;
    char             *bufptr = NULL;
    BOOL              bDone = FALSE;
    DWORD             objcount = 0;

    if (lpqos->ProviderSpecific.len == 0)
        return 0;

    bufptr = lpqos->ProviderSpecific.buf;
```

```
    objhdr = (QOS_OBJECT_HDR *)bufptr;


    while (!bDone)
    {
        if (objhdr->ObjectType == RSVP_OBJECT_STATUS_INFO)
        {
            status = (RSVP_STATUS_INFO *)objhdr;
            if (status->StatusCode & dwFlags)
                return 1;
        }
        else if (objhdr->ObjectType == QOS_OBJECT_END_OF_LIST)
            bDone = TRUE;

        bufptr += objhdr->ObjectLength;
        objcount += objhdr->ObjectLength;
        objhdr = (QOS_OBJECT_HDR *)bufptr;

        if (objcount >= lpqos->ProviderSpecific.len)
            bDone = TRUE;
    }
    return 0;
}

//
// Function: HandleClientEvents
//
// Description:
//    This function is called by the Server function to handle
//    events that occurred on client SOCKET handles. The socket
//    array is passed in along with the event array and the index
//    of the client who received the signal. Within the function,
//    the event is decoded and the appropriate action occurs.
//
void HandleClientEvents(SOCKET socks[], HANDLE events[], int index)
{
    WSANETWORKEVENTS  ne;
    char              databuf[4096];
    WSABUF            wbuf;
    DWORD             dwBytesRecv,
                      dwFlags;
    int               ret,
                      i;

    // Enumerate the network events that occurred
    //
    ret = WSAEnumNetworkEvents(socks[index], events[index], &ne);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAEnumNetworkEvents() failed: %d\n",
            WSAGetLastError());
        return;
    }
```

```
// Data to be read
//
if ((ne.lNetworkEvents & FD_READ) == FD_READ)
{
    wbuf.buf = databuf;
    wbuf.len = 4096;

    if (ne.iErrorCode[FD_READ_BIT])
        printf("FD_READ error: %d\n",
            ne.iErrorCode[FD_READ_BIT]);
    else
        printf("FD_READ\n");

    dwFlags = 0;
    ret = WSARecv(socks[index], &wbuf, 1, &dwBytesRecv,
        &dwFlags, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSARecv() failed: %d\n", WSAGetLastError());
        return;
    }
    wbuf.len = dwBytesRecv;

    printf("Read: %d bytes\n", dwBytesRecv);
}
// Able to write data; nothing to do here
//
if ((ne.lNetworkEvents & FD_WRITE) == FD_WRITE)
{
    if (ne.iErrorCode[FD_WRITE_BIT])

        printf("FD_WRITE error: %d\n",
            ne.iErrorCode[FD_WRITE_BIT]);
    else
        printf("FD_WRITE\n");
}
// The client closed the connection. Close the socket on our
// end and clean up the data structures.
//
if ((ne.lNetworkEvents & FD_CLOSE) == FD_CLOSE)
{
    if (ne.iErrorCode[FD_CLOSE_BIT])
        printf("FD_CLOSE error: %d\n",
            ne.iErrorCode[FD_CLOSE_BIT]);
    else
        printf("FD_CLOSE ...\n");
    closesocket(socks[index]);
    WSACloseEvent(events[index]);

    socks[index] = INVALID_SOCKET;
    //
    // Remote the client socket entry from the array and
    // compact the remaining clients to the beginning of the
```

```
    // array
    //
    for(i = index; i < MAX_CONN - 1; i++)
        socks[i] = socks[i + 1];
    nConns--;
}
// Received an FD_QOS event. This could mean several things.
//
if ((ne.lNetworkEvents & FD_QOS) == FD_QOS)
{
    char        buf[QOS_BUFFER_SZ];
    QOS        *lpqos = NULL;
    DWORD       dwBytes;

    if (ne.iErrorCode[FD_QOS_BIT])
        printf("FD_QOS error: %d\n",
            ne.iErrorCode[FD_QOS_BIT]);
    else
        printf("FD_QOS\n");

    lpqos = (QOS *)buf;
    lpqos->ProviderSpecific.buf = &buf[sizeof(QOS)];
    lpqos->ProviderSpecific.len = sizeof(buf) - sizeof(QOS);

    ret = WSAIoctl(socks[index], SIO_GET_QOS, NULL, 0,
            buf, QOS_BUFFER_SZ, &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl(SIO_GET_QOS) failed: %d\n",
            WSAGetLastError());
        return;
    }
    PrintQos(lpqos);
    //
    // See whether we're set for receiving FD_QOS events only.
    // If so, we need to invoke QOS on the connection
    // now; otherwise, client will never receive a RESV message.
    //
    if (iSetQos == SET_QOS_EVENT)
    {
        lpqos->ReceivingFlowspec.ServiceType =
            serverQos.ReceivingFlowspec.ServiceType;

        ret = WSAIoctl(socks[index], SIO_SET_QOS, lpqos,
            dwBytes, NULL, 0, &dwBytes, NULL, NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
                WSAGetLastError());
            return;
        }
        //
        // Change iSetQos so we don't set QOS again if we
```

```
            // receive another FD_QOS event
            //
            iSetQos = SET_QOS_BEFORE;
        }
    }
    return;
}

//
// Function: SrvCondAccept
//
// Description:
//    This is the conditional function for WSAAccept. The QOS service
//    provider is limited in that the QOS values passed into here are
//    unreliable, so the option SET_QOS_DURING is useless unless we call
//    SIO_SET_QOS with our own values (as opposed to the values the client
//    is requesting since those values are supposed to be returned in
//    lpSQOS). Note that on Windows 98, if lpSQOS is not NULL you have to
//    set some QOS values (with SIO_SET_QOS) in the conditional
//    function; otherwise, WSAAccept will fail.
//

int CALLBACK SrvCondAccept(LPWSABUF lpCallerId,
    LPWSABUF lpCallerdata, LPQOS lpSQOS, LPQOS lpGQOS,
    LPWSABUF lpCalleeId, LPWSABUF lpCalleeData, GROUP *g,
    DWORD dwCallbackData)
{
    DWORD       dwBytes = 0;
    SOCKET      s = (SOCKET)dwCallbackData;
    SOCKADDR_IN client;
    int         ret;

    if (nConns == MAX_CONN)
        return CF_REJECT;

    memcpy(&client, lpCallerId->buf, lpCallerId->len);
    printf("Client request: %s\n", inet_ntoa(client.sin_addr));

    if (iSetQos == SET_QOS_EVENT)
    {
        printf("Setting for event!\n");
        serverQos.SendingFlowspec.ServiceType |=
            SERVICE_NO_QOS_SIGNALING;
        serverQos.ReceivingFlowspec.ServiceType |=
            SERVICE_NO_QOS_SIGNALING;

        ret = WSAIoctl(s, SIO_SET_QOS, &serverQos,
            sizeof(serverQos), NULL, 0, &dwBytes, NULL, NULL);
        if (ret == SOCKET_ERROR)
        {
            printf("WSAIoctl() failed: %d\n",
                WSAGetLastError());
            return CF_REJECT;
```

```
        }
    }
    return CF_ACCEPT;
}

//
// Function: Server
//
// Description:
//    This server routine handles incoming client connections.
//    First it sets up the listening socket, sets QOS when
//    appropriate, and waits for incoming clients and events.
//
void Server(SOCKET s)
{
    SOCKET        sc[MAX_CONN + 1];
    WSAEVENT      hAllEvents[MAX_CONN+1];
    SOCKADDR_IN   local,
                  client;
    int           clientsz,
                  ret,
                  i;
    DWORD         dwBytesRet;
    WSANETWORKEVENTS ne;

    // Initialize the arrays to invalid values
    //
    for(i = 0; i < MAX_CONN+1; i++)
    {
        hAllEvents[i] = WSA_INVALID_EVENT;
        sc[i] = INVALID_SOCKET;
    }
    // Array index 0 will be our listening socket
    //
    hAllEvents[0] = WSACreateEvent();
    sc[0]         = s;
    nConns        = 0;

    local.sin_family = AF_INET;
    local.sin_port = htons(5150);
    local.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(s, (SOCKADDR *)&local, sizeof(local)) == SOCKET_ERROR)
    {
        printf("bind() failed: %d\n", WSAGetLastError());
        return;
    }
    listen(s, 7);

    if (iSetQos == SET_QOS_BEFORE)
    {
        ret = WSAIoctl(sc[0], SIO_SET_QOS, &serverQos,
            sizeof(serverQos), NULL, 0, &dwBytesRet, NULL, NULL);
```

```
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
            WSAGetLastError());
        return;
    }
    printf("Set QOS on listening socket:\n");
    PrintQos(&serverQos);
}


if (WSAEventSelect(sc[0], hAllEvents[0], FD_ACCEPT) ==
    SOCKET_ERROR)
{
    printf("WSAEventSelect() failed: %d\n", WSAGetLastError());
    return;
}

while (1)
{
    ret = WSAWaitForMultipleEvents(nConns+1, hAllEvents, FALSE,
        WSA_INFINITE, FALSE);
    if (ret == WSA_WAIT_FAILED)
    {
        printf("WSAWaitForMultipleObject() failed: %d\n",
            WSAGetLastError());
        return;
    }
    if ((i = ret - WSA_WAIT_EVENT_0) > 0)  // Client network event
        HandleClientEvents(sc, hAllEvents, i);
    else
    {
        ret = WSAEnumNetworkEvents(sc[0], hAllEvents[0],
            &ne);
        if (ret == SOCKET_ERROR)
        {
            printf("WSAEnumNetworkevents() failed: %d\n",
                WSAGetLastError());
            return;
        }
        if ((ne.lNetworkEvents & FD_ACCEPT) == FD_ACCEPT)
        {
            if (ne.iErrorCode[FD_ACCEPT_BIT])
                printf("FD_ACCEPT error: %d\n",
                    ne.iErrorCode[FD_ACCEPT_BIT]);
            else
                printf("FD_ACCEPT\n");

            clientsz = sizeof(client);
            sc[++nConns] = WSAAccept(s, (SOCKADDR *)&client,
                &clientsz, SrvCondAccept, sc[nConns]);
            if (sc[nConns] == SOCKET_ERROR)
            {
```

```
                    printf("WSAAccept() failed: %d\n",
                        WSAGetLastError());
                    nConns--;
                    return;
                }
                hAllEvents[nConns] = WSACreateEvent();

                Sleep(10000);
                if (iSetQos == SET_QOS_AFTER)
                {
                    ret = WSAIoctl(sc[nConns], SIO_SET_QOS,
                        &serverQos, sizeof(serverQos), NULL, 0,
                        &dwBytesRet, NULL, NULL);
                    if (ret == SOCKET_ERROR)
                    {
                        printf("WSAIoctl() failed: %d\n",
                            WSAGetLastError());
                        return;
                    }
                }
                ret = WSAEventSelect(sc[nConns],
                    hAllEvents[nConns], FD_READ | FD_WRITE |
                    FD_CLOSE | FD_QOS);
                if (ret == SOCKET_ERROR)
                {
                    printf("WSAEventSelect() failed: %d\n",
                        WSAGetLastError());
                    return;
                }
            }
            if (ne.lNetworkEvents & FD_CLOSE)
                printf("FD_CLOSEn");
            if (ne.lNetworkEvents & FD_READ)
                printf("FD_READn");
            if (ne.lNetworkEvents & FD_WRITE)
                printf("FD_WRITEn");
            if (ne.lNetworkEvents & FD_QOS)
                printf("FD_QOS\n");
        }
    }
    return;
}

//
// Function: Client
//
// Description:
//    The client routine initiates the connection, sets QOS when
//    appropriate, and handles incoming events.
//
void Client(SOCKET s)
{
    SOCKADDR_IN  server,
```

```
            local;
WSABUF      wbuf;
DWORD       dwBytes,
            dwBytesSent,
            dwBytesRecv,
            dwFlags;
HANDLE      hEvent;
int         ret, i;
char        databuf[DATA_BUFFER_SZ];
QOS        *lpqos;
WSANETWORKEVENTS ne;

hEvent = WSACreateEvent();
if (hEvent == NULL)
{
    printf("WSACreateEvent() failed: %d\n", WSAGetLastError());
    return;
}

lpqos = NULL;
if (iSetQos == SET_QOS_BEFORE)
{
    local.sin_family = AF_INET;
    local.sin_port = htons(0);
    local.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(s, (SOCKADDR *)&local, sizeof(local)) ==
        SOCKET_ERROR)
    {
        printf("bind() failed: %d\n", WSAGetLastError());
        return;
    }
    ret = WSAIoctl(s, SIO_SET_QOS, &clientQos,
        sizeof(clientQos), NULL, 0, &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
            WSAGetLastError());
        return;
    }
}
else if (iSetQos == SET_QOS_DURING)
    lpqos = &clientQos;
else if (iSetQos == SET_QOS_EVENT)
{
    clientQos.SendingFlowspec.ServiceType |=
        SERVICE_NO_QOS_SIGNALING;
    clientQos.ReceivingFlowspec.ServiceType |=
        SERVICE_NO_QOS_SIGNALING;

    ret = WSAIoctl(s, SIO_SET_QOS, &clientQos,
        sizeof(clientQos), NULL, 0, &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
```

```
    {
        printf("WSAIoctl() failed: %d\n", WSAGetLastError());
        return;
    }
}

server.sin_family = AF_INET;
server.sin_port = htons(5150);
server.sin_addr.s_addr = inet_addr(szServerAddr);

printf("Connecting to: %s\n", inet_ntoa(server.sin_addr));

ret = WSAConnect(s, (SOCKADDR *)&server, sizeof(server),
    NULL, NULL, lpqos, NULL);
if (ret == SOCKET_ERROR)
{
    printf("WSAConnect() failed: %d\n", WSAGetLastError());
    return;
}


ret = WSAEventSelect(s, hEvent, FD_READ | FD_WRITE |
    FD_CLOSE | FD_QOS);
if (ret == SOCKET_ERROR)
{
    printf("WSAEventSelect() failed: %d\n", WSAGetLastError());
    return;
}

wbuf.buf = databuf;
wbuf.len = DATA_BUFFER_SZ;

memset(databuf, '#', DATA_BUFFER_SZ);
databuf[DATA_BUFFER_SZ-1] = 0;

while (1)
{
    ret = WSAWaitForMultipleEvents(1, &hEvent, FALSE,
        WSA_INFINITE, FALSE);
    if (ret == WSA_WAIT_FAILED)
    {
        printf("WSAWaitForMultipleEvents() failed: %d\n",

            WSAGetLastError());
        return;
    }

    ret = WSAEnumNetworkEvents(s, hEvent, &ne);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAEnumNetworkEvents() failed: %d\n",
            WSAGetLastError());
        return;
```

```
}
if (ne.lNetworkEvents & FD_READ)
{
    if (ne.iErrorCode[FD_READ_BIT])
        printf("FD_READ error: %d\n",
            ne.iErrorCode[FD_READ_BIT]);
    else
        printf("FD_READ\n");

    wbuf.len = 4096;
    dwFlags = 0;
    ret = WSARecv(s, &wbuf, 1, &dwBytesRecv, &dwFlags,
        NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSARecv() failed: %d\n",
            WSAGetLastError());
        return;
    }
    printf("Read: %d bytes\n", dwBytesRecv);

    wbuf.len = dwBytesRecv;
    ret = WSASend(s, &wbuf, 1, &dwBytesSent, 0, NULL,
        NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSASend() failed: %d\n",
            WSAGetLastError());
        return;
    }
    printf("Sent: %d bytes\n", dwBytesSent);
}
if (ne.lNetworkEvents & FD_WRITE)
{
    if (ne.iErrorCode[FD_WRITE_BIT])
        printf("FD_WRITE error: %d\n",
            ne.iErrorCode[FD_WRITE_BIT]);
    else
        printf("FD_WRITE\n");

    if (!bWaitToSend)
    {
        wbuf.buf = databuf;
        wbuf.len = DATA_BUFFER_SZ;
        //
        // If the network can't support the bandwidth,
        // don't send
        //
        if (!AbleToSend(s))
        {
            printf("Network is unable to provide "
                "sufficient best-effort bandwidth\n");
            printf("before the reservation "
```

```
                "request is approved\n");
        }

        for(i = 0; i < 1; i++)
        {
            ret = WSASend(s, &wbuf, 1, &dwBytesSent, 0,
                NULL, NULL);
            if (ret == SOCKET_ERROR)
            {
                printf("WSASend() failed: %d\n",
                    WSAGetLastError());
                return;
            }
            printf("Sent: %d bytes\n", dwBytesSent);
        }
    }
}
if (ne.lNetworkEvents & FD_CLOSE)
{
    if (ne.iErrorCode[FD_CLOSE_BIT])
        printf("FD_CLOSE error: %d\n",
            ne.iErrorCode[FD_CLOSE_BIT]);
    else
        printf("FD_CLOSE ...\n");
    closesocket(s);
    WSACloseEvent(hEvent);
    return;
}
if (ne.lNetworkEvents & FD_QOS)
{
    char        buf[QOS_BUFFER_SZ];
    QOS        *lpqos = NULL;
    DWORD       dwBytes;
    BOOL        bRecvRESV = FALSE;

    if (ne.iErrorCode[FD_QOS_BIT])
    {
        printf("FD_QOS error: %d\n",
            ne.iErrorCode[FD_QOS_BIT]);
        if (ne.iErrorCode[FD_QOS_BIT] == WSA_QOS_RECEIVERS)
            bRecvRESV = TRUE;
    }
    else
        printf("FD_QOS\n");

    lpqos = (QOS *)buf;
    ret = WSAIoctl(s, SIO_GET_QOS, NULL, 0,
            buf, QOS_BUFFER_SZ, &dwBytes, NULL, NULL);
    if (ret == SOCKET_ERROR)
    {
        printf("WSAIoctl(SIO_GET_QOS) failed: %d\n",
            WSAGetLastError());
        return;
```

```
            }
            PrintQos(lpqos);
            //
            // Check to see whether a status object is returned
            // in the QOS structure that might also contain the
            // WSA_QOS_RECEIVERS flag
            //
            if (ChkForQosStatus(lpqos, WSA_QOS_RECEIVERS))
                bRecvRESV = TRUE;

            if (iSetQos == SET_QOS_EVENT)
            {
                lpqos->SendingFlowspec.ServiceType =
                    clientQos.SendingFlowspec.ServiceType;
                ret = WSAIoctl(s, SIO_SET_QOS, lpqos, dwBytes,
                        NULL, 0, &dwBytes, NULL, NULL);
                if (ret == SOCKET_ERROR)
                {
                    printf("WSAIoctl(SIO_SET_QOS) failed: %d\n",
                        WSAGetLastError());
                    return;
                }
                //
                // Change iSetQos so that we don't set QOS again if we
                // receive another FD_QOS event
                //
                iSetQos = SET_QOS_BEFORE;
            }

            if (bWaitToSend && bRecvRESV)
            {
                wbuf.buf = databuf;
                wbuf.len = DATA_BUFFER_SZ;

                for(i = 0; i < 1; i++)
                {
                    ret = WSASend(s, &wbuf, 1, &dwBytesSent, 0,
                        NULL, NULL);
                    if (ret == SOCKET_ERROR)
                    {
                        printf("WSASend() failed: %d\n",
                            WSAGetLastError());
                        return;
                    }
                    printf("Sent: %d bytes\n", dwBytesSent);
                }
            }
        }
    }
    return;
}

//
```

```
// Function: main
//
// Description:
//    Initialize Winsock, parse command line arguments, create
//    a QOS TCP socket, and call the appropriate handler
//    routine depending on the arguments supplied
//
int main(int argc, char **argv)
{
    WSADATA           wsd;
    WSAPROTOCOL_INFO *pinfo = NULL;
    SOCKET            s;

    // Parse the command line
    ValidateArgs(argc, argv);
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("Unable to load Winsock: %d\n", GetLastError());
        return -1;
    }
    pinfo = FindProtocolInfo(AF_INET, SOCK_STREAM, IPPROTO_TCP,
        XP1_QOS_SUPPORTED);
    if (!pinfo)
    {
        printf("unable to find suitable provider!\n");
        return -1;
    }
    printf("Provider returned: %s\n", pinfo->szProtocol);

    s = WSASocket(FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
        FROM_PROTOCOL_INFO, pinfo, 0, WSA_FLAG_OVERLAPPED);
    if (s == INVALID_SOCKET)
    {
        printf("WSASocket() failed: %d\n", WSAGetLastError());
        return -1;
    }
    InitQos();

    if (bServer)
        Server(s);
    else
        Client(s);

    closesocket(s);
    WSACleanup();
    return 0;
}
```

Table 12-8. *Qostcp.c command line parameters*

-q:[b,d,a,e]	Set QOS either before (b), during (d), or after (a), or wait for an *FD_QOS* event (e) before setting
-s	

Act as the server
-c:Server-IP
Act as the client, and connect to the server at the given IP address
-w
Wait to send data until a RESV message has been received
-r
Set the option to receive notification once the reservation has been confirmed

Parameter	*Meaning*

Unicast UDP

Using unicast UDP offers a few more possibilities than TCP does. The UDP example on the accompanying CD is named *Qosudp.c.* It incorporates both the sender and the receiver. For the sender, there are two methods for telling the QOS service provider where data is to be sent. Remember that the peer's address is required to initiate an RSVP session. This can be accomplished by calling either *WSAConnect* or *SIO_SET_QOS* with the *QOS_DESTADDR* object. The unicast UDP example also takes a parameter on when QOS is to be set. If the user specifies that QOS is to be set before binding or connecting, the *SIO_SET_QOS* ioctl command with a *QOS_DESTADDR* object is used. If the user specifies that QOS is to be set during *WSAAccept* 's conditional function, QOS parameters are specified in the *WSAConnect* call. If the user specifies that QOS is to be set after session setup, *WSAConnect* is called without QOS and *SIO_SET_QOS* is invoked later without the *QOS_DESTADDR* object. Finally, if the user specifies that QOS is to be set only upon receipt of a *FD_QOS* event notification, *SIO_SET_QOS* is called with a *QOS_DESTADDR* object, but the flag *SERVICE_QOS_NO_SIGNALING* is ORed in with the *ServiceType FLOWSPEC* field.

The receiving side has considerably fewer options. The various flags for when to set QOS don't really apply here. QOS can be set before data is to be received, or the receiver can wait for an *FD_QOS* event to occur. This is because UDP doesn't receive any connection request—there is no setting QOS during the accept function or after session setup. The receiver also has the option of specifying a different filter style, such as fixed filter or shared explicit. If a different filter is specified, an IP address must be given with the -r:IP option. The function *SetQosReceivers* fills in an *RSVP_RESERVE_INFO* structure with an *RSVP_FILTERSPEC* structure, which defines the sender's IP address. One particularly important aspect of setting the filter is that the port number of the sender must be specified. This means that the receiver must know each sender's IP address and the port number it is bound to on the sender's side.

Note that the receiver can also use *WSAConnect* to associate the sender's IP address with the socket. However, because the UDP receiver can specify various filter styles as well as a number of senders, *WSAConnect* cannot be used. Remember that if *WSAConnect* is used to associate the endpoint's IP address, send and receive operations can occur only with that peer and have QOS associated with it.

If you compare the unicast UDP example to the TCP example, you'll find they're quite similar. The only exception is the different way QOS can be set on the socket. UDP applications require that the sender specify the recipient's IP address to invoke RSVP, and QOS provides two methods for this. TCP applications by their nature do this by default in the connect call. The event loop for both applications is almost the same. Don't forget the one major "gotcha" for UDP applications: the socket must be bound locally before any QOS (sending or receiving) can be set on it with *SIO_SET_QOS* when *WSAConnect* is not used. Binding to *INADDR_ANY* and port 0 is perfectly legal, as well as using a specific IP and port. Using *WSAConnect* performs an implicit bind, so if QOS is set at that point, you do not have to explicitly bind beforehand.

Multicast UDP

The last example provided is multicast QOS, which can be found on the CD as Qosmcast.c. The central function here is *WSAJoinLeaf* , which an application must call to join a multicast group, as we saw in the previous chapter . When an application joins the group, it can also pass QOS parameters. The multicast example takes the same parameters as the unicast UDP example. You can select the point at which QOS can be set on the socket. If you choose to set QOS during the accept conditional function, QOS is passed into the *WSAJoinLeaf* call. Otherwise, set QOS by calling *WSAIoctl* with *SIO_SET_QOS* .

One of the parameters for receivers allows users to set the filter style to either fixed filter or shared explicit. Remember that multicast UDP uses wildcard style by default. Specifying a different filter type applies only to receivers, and if either of these two filter types is desired, each sender's address is specified in an -r:SenderIP command line option. Users can specify the filters via the -f option with se for shared explicit and with ff for fixed filter.

For both sender and receiver, the -m option is used to specify the multicast group to join. Note that this option

can be specified multiple times to join as many groups as you want. The -s option indicates that the program should act as a sender. The w option tells the sender to wait until there is a *WSA_QOS_RECEIVERS* notification before sending any data. Finally, the -q option identifies when to set QOS. No matter when QOS is specified to be set, the socket is bound locally to port 5150. In actuality, any port can be chosen or 0 can be specified so that the port is chosen for you; however, if the receiver is to set either the fixed or the shared explicit filter, it must also provide the IP and port of the sender. We use a fixed port for the sake of simplicity. Unlike in unicast UDP, the receiver is not *required* to bind the port locally to set QOS. The reason for this is that *WSAJoinLeaf* implicitly binds the socket if it is not already bound. The other question you might be asking is whether the socket option commands *IP_ ADD_MEMBERSHIP* and *IP_ DROP_MEMBERSHIP* can be used instead of *WSAJoinLeaf* . The answer is no. If these commands are used, the QOS parameters requested will not be applied to the socket.

We won't go into the exact details of what happens when QOS is chosen to be set because it is quite similar to the unicast UDP example. By now, you should be familiar with how the RSVP session is initiated and what is needed to generate the PATH and RESV messages.

ATM and QOS

As we mentioned earlier, QOS is natively available on ATM networks. Both Windows 2000 and Windows 98 (with Service Pack 1) support native ATM programming from Winsock, as you learned in Chapter 6. QOS is natively available on an ATM network, which means that the network, application, and policy components that are necessary for QOS over IP are not required over ATM. This includes the Admission Control Service and the RSVP protocol. Instead, the ATM switch performs bandwidth allocations and prevents over-allocation of bandwidth.

In addition to the differences we've already mentioned, the Winsock API functions behave a bit differently with ATM QOS than they do with QOS over IP. The first major difference is that the QOS bandwidth request is handled as part of the connection request. This differs from QOS over IP in that the RSVP session is established separately from the connection. Also, if the bandwidth request is rejected under ATM, the connection will fail.

This leads to our next point: both of the native ATM providers are connection-oriented. As a result, you don't have the problem of setting QOS levels for a connectionless socket and then having to specify the endpoint for communication. The next major difference is that only one side sets the QOS parameters for a connection. That is, if the client wants to set QOS on a connection, both the sending and receiving *FLOWSPEC* structures are set within the QOS structure passed to *WSAConnect*. These values will then be applied to the connection, in contrast to QOS over IP, in which the sender requests certain QOS levels and the receiver then makes the actual reservation. Additionally, the listening socket might have QOS set using *WSAIoctl* and *SIO_SET_QOS*. These values will be applied to any incoming connections. This also means that QOS *must* be set during connection setup. You cannot set QOS on an already established connection.

This leads us to our last point: once QOS is set for a connection, you cannot renegotiate QOS by calling *WSAIoctl* and *SIO_SET_QOS*. When QOS is set on a connection, it remains until the connection is closed.

Keep in mind that RSVP is not present and no signaling occurs. This means that none of the status flags in Table 12-6 are ever generated. QOS is set when establishing the connection, and no further notifications or events occur until the connection is closed.

Conclusion

QOS offers powerful capabilities to applications that require a guaranteed level of network service. Setting up a QOS connection is rather involved, but don't let this scare you. The most important concept is learning how and when RSVP messages are generated so that you can code your application accordingly.

Chapter 13

Raw Sockets

A raw socket is a socket that allows access to the underlying transport protocol. This chapter is dedicated to illustrating how raw sockets can be used to simulate IP utilities, such as Traceroute and Ping. Raw sockets can also be used to actually manipulate the IP header information. This chapter is concerned only with the IP protocol; we will not address raw sockets with any other protocol, as most protocols (except ATM) do not support raw sockets at all. All raw sockets are created using the *SOCK_RAW* socket type and are currently supported only under Winsock 2. Therefore, neither Microsoft Windows CE nor Windows 95 (without the Winsock 2 update) can utilize raw sockets.

Additionally, using raw sockets requires substantial knowledge of the underlying protocol structure, which is generally not the focus of this book. In this chapter, we will discuss the Internet Control Message Protocol (ICMP), the Internet Group Management Protocol (IGMP), and the User Datagram Protocol (UDP). ICMP is used by the Ping utility, which can detect whether a route to a host is valid and whether the host machine is responding. Developers often need a programmatic method of determining whether a machine is alive and reachable. IGMP is used by IP multicasting to advertise multicast group membership to routers. Recently support was added to most Win32 platforms to support IGMP version 2. However, in some cases you might want to send your own IGMP packets to drop group membership. We will examine the UDP protocol in conjunction with the *IP_HDRINCL* socket option as an example of how to send your own IGMP packets. For all three of these protocols, we will cover only the aspects necessary to fully explain the code in this chapter and in the example programs. For more detailed information, consult W. Richard Stevens's book on IP, *TCP/IP Illustrated Vol. 1* (Addison-Wesley, 1994).

Raw Socket Creation

The first step in using raw sockets is creating the socket. You can use either *socket* or *WSASocket*. Note that typically, no catalog entry in Winsock for IP has the *SOCK_RAW* socket type. However, this does not prevent you from creating this type of socket. It just means that you cannot create a raw socket using a *WSAPROTOCOL_INFO* structure. Refer back to Chapter 5 for information on enumerating protocol entries with the *WSAEnumProtocols* function and the *WSAPROTOCOL_INFO* structure. You must specify the *SOCK_RAW* flag yourself in socket creation. The following code snippet illustrates the creation of a raw socket using ICMP as the underlying IP protocol.

```
SOCKET    s;

s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
// Or
s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_ICMP, NULL, 0,
    WSA_FLAG_OVERLAPPED);
if (s == INVALID_SOCKET)
{
    // Socket creation failed
}
```

Because raw sockets offer the ability to manipulate the underlying transport, they can be used for malicious purposes and are a security issue in Windows NT. Therefore, only members of the Administrators group can create sockets of type *SOCK_RAW*. Windows 95 and Windows 98 do not impose any kind of limitation.

To work around this problem in Windows NT, you can disable the security check on raw sockets by creating the following registry variable and setting its value to the integer 1 as a *DWORD* type.

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet
    \Services\Afd\Parameters\DisableRawSecurity
```

After the registry change, you need to reboot the machine.

In the above code example, we used the ICMP protocol, but you can also use IGMP, UDP, IP, or raw IP using the flags *IPPROTO_IGMP*, *IPPROTO_UDP*, *IPPROTO_IP*, or *IPPROTO_RAW*, respectively. However, be aware of one limitation: On Windows NT 4, Windows 98, and Windows 95 (with Winsock 2), you can use only IGMP and ICMP when creating raw sockets. The protocol flags *IPPROTO_UDP*, *IPPROTO_IP*, and *IPPROTO_RAW* require the use of the socket option *IP_HDRINCL*, which is not supported on those platforms. Windows 2000 does, however, support *IP_HDRINCL*, so it is possible to manipulate the IP header itself (*IPPROTO_RAW*), the TCP header (*IPPROTO_TCP*), and the UDP header (*IPPROTO_UDP*).

Once the raw socket is created with the appropriate protocol flags, you can use the socket handle in send and receive calls. When creating raw sockets, the IP header will be included in the data returned upon any receive, regardless of whether the *IP_HDRINCL* option is set.

Internet Control Message Protocol

ICMP is used as a means of messaging between hosts. Most ICMP messages relate to errors that occur in communication between hosts; the remaining ICMP messages are used to query hosts. The ICMP protocol uses IP addressing because it is a protocol encapsulated within an IP datagram. Figure 13-1 illustrates the fields of an ICMP message. The ICMP message is wrapped in an IP header.

The first field is the ICMP message type, which can be classified as either a query or an error. The code field further defines the type of query or message. The checksum field is the 16-bit one's complement sum of the ICMP header. Finally, the ICMP contents depend on the ICMP type and code. Table 13-1 lists the various types and codes.

When an ICMP error message is generated, the message always contains the IP header and the first 8 bytes of the IP datagram that caused the ICMP error to occur. This allows the host receiving the ICMP error to associate the message with one particular protocol and process associated with that error. In our case, Ping relies on the echo request and echo reply ICMP queries rather than on error messages. Hosts generate ICMP messages in response to problems with TCP or UDP; ICMP doesn't have many applications beyond that. In the next section, we will discuss how to use the ICMP protocol with a raw socket to generate a Ping request by using the echo request and echo reply messages. If you require additional information about ICMP errors or the other types of ICMP queries, consult more in-depth sources, such as Stevens's *TCP/IP Illustrated Vol. 1*.

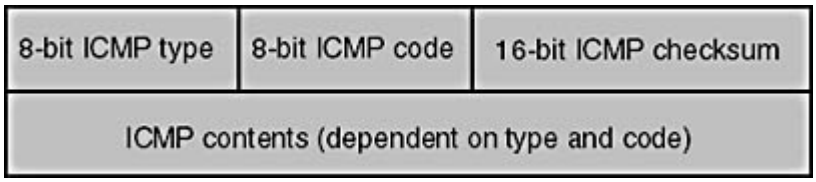

Figure 13-1. *ICMP header*

Table 13-1. *ICMP message types*

Type	*Query/Error (Error Type)*	*Code*	*Description*
0	Query	0	Echo reply
3	Error: Destination unreachable	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	Network unreachable Host unreachable Protocol unreachable Port unreachable Fragmentation needed, but the Don't Fragment bit has been set Source route failed Destination network unknown Destination host unknown Source host isolated (obsolete) Destination network administratively prohibited Destination host administratively prohibited Network unreachable for TOS Host unreachable for TOS Communication administratively prohibited by filtering Host precedence violation Precedence cutoff in effect
4	Error	0	Source quench
5	Error: Redirect	0 1 2 3	Redirect for network Redirect for host Redirect for TOS and network Redirect for TOS and host
8	Query	0	Echo request
9	Query	0	Router advertisement

10	Query	0	Router solicitation
11	Error: Time exceeded	0 1	TTL equals 0 during transit TTL equals 0 during reassembly
12	Error: Parameter problem	0	IP header bad Required option missing
13	Query	0	Time stamp request
14	Query	0	Time stamp reply
15	Query	0	Information request
16	Query	0	Information reply
17	Query	0	Address mask request
18	Query	0	Address mask reply

Ping Example

A ping is often used to determine whether a particular host is alive and reachable through the network. By generating an ICMP echo request and directing it to the host you are interested in, you can determine whether you can successfully reach that machine. Of course, this does not guarantee that a socket client will be able to connect to a process on that host (a process on the remote server might not be listening, for example); it just means that the network layer of the remote host is responding to network events. Essentially, the Ping example performs the following steps:

1. Creates a socket of type *SOCK_RAW* and protocol *IPPROTO_ICMP*
2. Creates and initializes the ICMP header
3. Calls *sendto* or *WSASendto* to send the ICMP request to the remote host
4. Calls *recvfrom* or *WSARecvfrom* to receive any ICMP responses

When you send the ICMP echo request, the remote machine intercepts the ICMP query and generates an echo reply message back to you. If for some reason the host is not reachable, the appropriate ICMP error message—such as destination host unreachable—will be returned by a router somewhere along the path to the intended recipient. If the physical network connection to the host is good but the remote host is either down or not responding to network events, you need to perform your own timeout to determine this. The Ping.c example in Figure 13-2 illustrates how to create a socket capable of sending and receiving ICMP packets, as well as how to use the *IP_OPTIONS* socket option to implement the record route option.

Figure 13-2. *Ping.c*

```
// Module Name: Ping.c
//
// Command Line Options/Parameters:
//     Ping [host] [packet-size]
//
//     host         String name of host to ping
//     packet-size  Integer size of packet to send
//                  (smaller than 1024 bytes)
//
//#pragma pack(1)

#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
```

```
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

#define IP_RECORD_ROUTE  0x7
//
// IP header structure
//
typedef struct _iphdr
{
    unsigned int   h_len:4;        // Length of the header
    unsigned int   version:4;      // Version of IP
    unsigned char  tos;            // Type of service
    unsigned short total_len;      // Total length of the packet
    unsigned short ident;          // Unique identifier
    unsigned short frag_and_flags; // Flags
    unsigned char  ttl;            // Time to live
    unsigned char  proto;          // Protocol (TCP, UDP, etc.)
    unsigned short checksum;       // IP checksum

    unsigned int   sourceIP;
    unsigned int   destIP;
} IpHeader;

#define ICMP_ECHO        8
#define ICMP_ECHOREPLY   0
#define ICMP_MIN         8 // Minimum 8-byte ICMP packet (header)

//
// ICMP header structure
//
typedef struct _icmphdr
{
    BYTE   i_type;
    BYTE   i_code;                 // Type sub code
    USHORT i_cksum;
    USHORT i_id;
    USHORT i_seq;
    // This is not the standard header, but we reserve space for time
    ULONG  timestamp;
} IcmpHeader;

//
// IP option header--use with socket option IP_OPTIONS
//
typedef struct _ipoptionhdr
{
    unsigned char        code;        // Option type
    unsigned char        len;         // Length of option hdr
    unsigned char        ptr;         // Offset into options
    unsigned long        addr[9];     // List of IP addrs
} IpOptionHeader;
```

```
#define DEF_PACKET_SIZE  32        // Default packet size
#define MAX_PACKET       1024      // Max ICMP packet size
#define MAX_IP_HDR_SIZE  60        // Max IP header size w/options

BOOL  bRecordRoute;
int   datasize;
char *lpdest;

//
// Function: usage
//
// Description:
//    Print usage information
//
void usage(char *progname)
{
    printf("usage: ping -r <host> [data size]\n");
    printf("       -r           record route\n");
    printf("        host        remote machine to Ping\n");
    printf("        datasize    can be up to 1 KB\n");
    ExitProcess(-1);
}

//
// Function: FillICMPData
//
// Description:
//    Helper function to fill in various fields for our ICMP request
//
void FillICMPData(char *icmp_data, int datasize)
{
    IcmpHeader *icmp_hdr = NULL;
    char       *datapart = NULL;

    icmp_hdr = (IcmpHeader*)icmp_data;
    icmp_hdr->i_type = ICMP_ECHO;        // Request an ICMP echo
    icmp_hdr->i_code = 0;
    icmp_hdr->i_id = (USHORT)GetCurrentProcessId();
    icmp_hdr->i_cksum = 0;
    icmp_hdr->i_seq = 0;

    datapart = icmp_data + sizeof(IcmpHeader);
    //
    // Place some junk in the buffer
    //
    memset(datapart,'E', datasize - sizeof(IcmpHeader));
}

//
// Function: checksum
//
// Description:
//    This function calculates the 16-bit one's complement sum
```

```
//    of the supplied buffer (ICMP) header
//
USHORT checksum(USHORT *buffer, int size)
{
    unsigned long cksum=0;

    while (size > 1)
    {
        cksum += *buffer++;
        size -= sizeof(USHORT);
    }
    if (size)
    {
        cksum += *(UCHAR*)buffer;
    }
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >>16);
    return (USHORT)(~cksum);
}

//
// Function: DecodeIPOptions
//
// Description:
//    If the IP option header is present, find the IP options
//    within the IP header and print the record route option
//    values
//
void DecodeIPOptions(char *buf, int bytes)
{
    IpOptionHeader *ipopt = NULL;
    IN_ADDR         inaddr;
    int             i;
    HOSTENT        *host = NULL;

    ipopt = (IpOptionHeader *)(buf + 20);

    printf("RR:   ");
    for(i = 0; i < (ipopt->ptr / 4) - 1; i++)
    {
        inaddr.S_un.S_addr = ipopt->addr[i];
        if (i != 0)
            printf("      ");
        host = gethostbyaddr((char *)&inaddr.S_un.S_addr,
                    sizeof(inaddr.S_un.S_addr), AF_INET);
        if (host)
            printf("(%-15s) %s\n", inet_ntoa(inaddr), host->h_name);
        else
            printf("(%-15s)\n", inet_ntoa(inaddr));
    }
    return;
}
```

```
//
// Function: DecodeICMPHeader
//
// Description:
//    The response is an IP packet. We must decode the IP header to
//    locate the ICMP data.
//
void DecodeICMPHeader(char *buf, int bytes,
    struct sockaddr_in *from)
{
    IpHeader       *iphdr = NULL;
    IcmpHeader     *icmphdr = NULL;
    unsigned short  iphdrlen;
    DWORD           tick;
    static   int    icmpcount = 0;

    iphdr = (IpHeader *)buf;
    // Number of 32-bit words * 4 = bytes
    iphdrlen = iphdr->h_len * 4;
    tick = GetTickCount();

    if ((iphdrlen == MAX_IP_HDR_SIZE) && (!icmpcount))
        DecodeIPOptions(buf, bytes);

    if (bytes  < iphdrlen + ICMP_MIN)
    {
        printf("Too few bytes from %s\n",
            inet_ntoa(from->sin_addr));
    }
    icmphdr = (IcmpHeader*)(buf + iphdrlen);

    if (icmphdr->i_type != ICMP_ECHOREPLY)
    {
        printf("nonecho type %d recvd\n", icmphdr->i_type);
        return;
    }

    // Make sure this is an ICMP reply to something we sent!
    //
    if (icmphdr->i_id != (USHORT)GetCurrentProcessId())
    {
        printf("someone else's packet!\n");
        return ;
    }
    printf("%d bytes from %s:", bytes, inet_ntoa(from->sin_addr));
    printf(" icmp_seq = %d. ", icmphdr->i_seq);
    printf(" time: %d ms", tick - icmphdr->timestamp);
    printf("\n");

    icmpcount++;
    return;
}
```

```
void ValidateArgs(int argc, char **argv)
{
    int                i;

    bRecordRoute = FALSE;
    lpdest = NULL;
    datasize = DEF_PACKET_SIZE;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'r':        // Record route option
                    bRecordRoute = TRUE;
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
        else if (isdigit(argv[i][0]))
            datasize = atoi(argv[i]);
        else
            lpdest = argv[i];
    }
}

//
// Function: main
//
// Description:
//    Set up the ICMP raw socket, and create the ICMP header. Add
//    the appropriate IP option header, and start sending ICMP
//    echo requests to the endpoint. For each send and receive,
//    we set a timeout value so that we don't wait forever for a
//    response in case the endpoint is not responding. When we
//    receive a packet, decode it.
//
int main(int argc, char **argv)
{

    WSADATA            wsaData;
    SOCKET             sockRaw = INVALID_SOCKET;
    struct sockaddr_in dest,
                       from;
    int                bread,
                       fromlen = sizeof(from),
                       timeout = 1000,
                       ret;
    char              *icmp_data = NULL,
                      *recvbuf = NULL;
```

```
    unsigned int       addr = 0;
    USHORT             seq_no = 0;
    struct hostent    *hp = NULL;
    IpOptionHeader     ipopt;


    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        printf("WSAStartup() failed: %d\n", GetLastError());
        return -1;
    }
    ValidateArgs(argc, argv);

    //
    // WSA_FLAG_OVERLAPPED flag is required for SO_RCVTIMEO,
    // SO_SNDTIMEO option. If NULL is used as last param for
    // WSASocket, all I/O on the socket is synchronous, the
    // internal user mode wait code never gets a chance to
    // execute, and therefore kernel-mode I/O blocks forever.
    // A socket created via the socket function has the over-
    // lapped I/O attribute set internally. But here we need
    // to use WSASocket to specify a raw socket.
    //
    // If you want to use timeout with a synchronous
    // nonoverlapped socket created by WSASocket with last
    // param set to NULL, you can set the timeout by using
    // the select function, or you can use WSAEventSelect and
    // set the timeout in the WSAWaitForMultipleEvents
    // function.
    //
    sockRaw = WSASocket (AF_INET, SOCK_RAW, IPPROTO_ICMP, NULL, 0,
                         WSA_FLAG_OVERLAPPED);
    if (sockRaw == INVALID_SOCKET)
    {
        printf("WSASocket() failed: %d\n", WSAGetLastError());
        return -1;
    }
    if (bRecordRoute)
    {
        // Setup the IP option header to go out on every ICMP packet
        //
        ZeroMemory(&ipopt, sizeof(ipopt));
        ipopt.code = IP_RECORD_ROUTE; // Record route option
        ipopt.ptr  = 4;               // Point to the first addr offset
        ipopt.len  = 39;              // Length of option header

        ret = setsockopt(sockRaw, IPPROTO_IP, IP_OPTIONS,
            (char *)&ipopt, sizeof(ipopt));
        if (ret == SOCKET_ERROR)
        {
            printf("setsockopt(IP_OPTIONS) failed: %d\n",
                WSAGetLastError());
        }
```

```
    }
    // Set the send/recv timeout values
    //
    bread = setsockopt(sockRaw, SOL_SOCKET, SO_RCVTIMEO,
                (char*)&timeout, sizeof(timeout));
    if(bread == SOCKET_ERROR)
    {
        printf("setsockopt(SO_RCVTIMEO) failed: %d\n",
            WSAGetLastError());
        return -1;
    }
    timeout = 1000;
    bread = setsockopt(sockRaw, SOL_SOCKET, SO_SNDTIMEO,
                (char*)&timeout, sizeof(timeout));
    if (bread == SOCKET_ERROR)
    {
        printf("setsockopt(SO_SNDTIMEO) failed: %d\n",
            WSAGetLastError());
        return -1;
    }
    memset(&dest, 0, sizeof(dest));
    //
    // Resolve the endpoint's name if necessary
    //
    dest.sin_family = AF_INET;
    if ((dest.sin_addr.s_addr = inet_addr(lpdest)) == INADDR_NONE)
    {
        if ((hp = gethostbyname(lpdest)) != NULL)
        {
            memcpy(&(dest.sin_addr), hp->h_addr, hp->h_length);
            dest.sin_family = hp->h_addrtype;
            printf("dest.sin_addr = %s\n", inet_ntoa(dest.sin_addr));
        }
        else
        {
            printf("gethostbyname() failed: %d\n",
                WSAGetLastError());
            return -1;
        }
    }

    //
    // Create the ICMP packet
    //
    datasize += sizeof(IcmpHeader);

    icmp_data = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                  MAX_PACKET);
    recvbuf = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                  MAX_PACKET);
    if (!icmp_data)
    {
        printf("HeapAlloc() failed: %d\n", GetLastError());
```

```
        return -1;
    }
    memset(icmp_data,0,MAX_PACKET);
    FillICMPData(icmp_data,datasize);
    //
    // Start sending/receiving ICMP packets
    //
    while(1)
    {
        static int nCount = 0;
        int        bwrote;

        if (nCount++ == 4)
            break;

        ((IcmpHeader*)icmp_data)->i_cksum = 0;
        ((IcmpHeader*)icmp_data)->timestamp = GetTickCount();
        ((IcmpHeader*)icmp_data)->i_seq = seq_no++;
        ((IcmpHeader*)icmp_data)->i_cksum =
            checksum((USHORT*)icmp_data, datasize);

        bwrote = sendto(sockRaw, icmp_data, datasize, 0,
                    (struct sockaddr*)&dest, sizeof(dest));
        if (bwrote == SOCKET_ERROR)
        {
            if (WSAGetLastError() == WSAETIMEDOUT)
            {
                printf("timed out\n");
                continue;
            }
            printf("sendto() failed: %d\n", WSAGetLastError());
            return -1;
        }
        if (bwrote < datasize)
        {
            printf("Wrote %d bytes\n", bwrote);
        }
        bread = recvfrom(sockRaw, recvbuf, MAX_PACKET, 0,
                    (struct sockaddr*)&from, &fromlen);
        if (bread == SOCKET_ERROR)
        {
            if (WSAGetLastError() == WSAETIMEDOUT)
            {
                printf("timed out\n");
                continue;
            }
            printf("recvfrom() failed: %d\n", WSAGetLastError());
            return -1;
        }
        DecodeICMPHeader(recvbuf, bread, &from);

        Sleep(1000);
    }
```

```
    // Cleanup
    //
    if (sockRaw != INVALID_SOCKET)
        closesocket(sockRaw);
    HeapFree(GetProcessHeap(), 0, recvbuf);
    HeapFree(GetProcessHeap(), 0, icmp_data);

    WSACleanup();
    return 0;
}
```

One noticeable feature of the Ping example is its use of the *IP_OPTIONS* socket option. We use the record route IP option so that when our ICMP packet hits a router, its IP address is added into the IP option header at the location indicated by the offset field in the IP option header. This offset is also incremented by 4 each time a router adds its address. The increment value is based on the fact that an IP version 4 address is 4 bytes in length. This book does not address any IP version 6 concerns, as no current Windows platforms support this yet. Once you receive the echo reply, decode the option header and print the IP addresses and host names of the routers visited. See Chapter 9 for more information on the other types of IP options available.

Traceroute

Another valuable IP networking tool is the Traceroute utility. This allows you to determine the IP addresses of the routers that are traversed in order to reach a certain host on the network. With Ping, using the record route option in the IP option header also allows you to determine the IP addresses of intermediary routers, but Ping is limited to only 9 hops—the maximum space allocated for addresses in the option header. A hop occurs whenever an IP datagram must pass through a router in order to traverse multiple physical networks. For routes with more than 9 hops, use Traceroute.

The idea behind Traceroute is to send a UDP packet to the destination and incrementally change the IP time-to-live (TTL) value. Initially, the TTL value is 1, which means the UDP packet will reach the first router, where the TTL will expire. The expiration will cause the router to generate an ICMP time-exceeded packet. Then the initial TTL value increases by 1, so this time the UDP packet gets one router farther and an ICMP time-exceeded packet is sent from that router. Collecting each of the ICMP messages gives you a clear path of the IP addresses traversed in order to reach the endpoint. Once the TTL is incremented enough so that packets actually reach the endpoint in question, an ICMP port-unreachable message is most likely returned, as no process on the recipient is waiting for this message.

Traceroute is a useful utility because it gives you a lot of information about the route to a particular host, which is often a concern when you use multicasting or when you experience routing problems. Fewer applications need to perform a Traceroute programmatically than a ping, but certain tasks might require Traceroute-like capabilities.

Two methods can be used to implement the Traceroute program. First you can use UDP packets and send datagrams, incrementally changing the TTL. Each time the TTL expires, an ICMP message will be returned to you. This method requires one socket of the UDP protocol to send the messages and another socket of the ICMP protocol to read the returned messages. The UDP socket is a normal UDP socket, as you saw in Chapter 7. The ICMP socket is of type *SOCK_RAW* and protocol *IPPROTO_ICMP*. The TTL of the UDP socket needs to be manipulated via the *IP_TTL* socket option. Alternatively, you can create a UDP socket and use the *IP_HDRINCL* option (discussed later in this chapter) to set the TTL manually within the IP header, but this is quite a lot of work.

The other method is simply to send ICMP packets to the destination, also incrementally changing the TTL. This also results in ICMP error messages being returned when the TTL expires. This method resembles the Ping example in that it requires only one socket (of the ICMP protocol). Under the sample code folder on the companion CD, you will find a Traceroute example using ICMP packets named Traceroute.c. We won't include the whole example in this chapter, as it is similar in design to the Ping example.

Internet Group Management Protocol

The IGMP protocol is used by IP multicasting to manage membership to multicast groups. Consult Chapter 11 for information on how to use Winsock to join and leave multicast groups. When an application joins a multicast group, it sends an IGMP message to every router on a local network, using the special address 224.0.0.2. This address is known as the all-routers group. This IGMP message tells the routers that there is a recipient for any data destined for the given multicast address. The routers forward data for this multicast group if and only if a recipient is interested in receiving that data. Without this ability, multicast data is nothing more than a broadcast.

To make matters a bit more confusing, there are two versions of the IGMP protocol: version 1 and version 2. They are described in RFC 1112 and RFC 2236, respectively. The major difference between them is that version 1 has no method for a host to tell the router to stop forwarding data destined for a multicast group. In other words, when a host decides to drop group membership, the host does not notify the router and the router continues to forward data for that group until it sends a group query for that group and no one responds. Version 2 of the protocol adds an explicit "leave" message so hosts can notify routers immediately when they are dropping membership. Additionally, the header formats for the two versions are slightly different. Figure 13-3 illustrates the version 1 header, while Figure 13-4 shows the version 2 header. At only 8 bytes in length, both headers are simple.

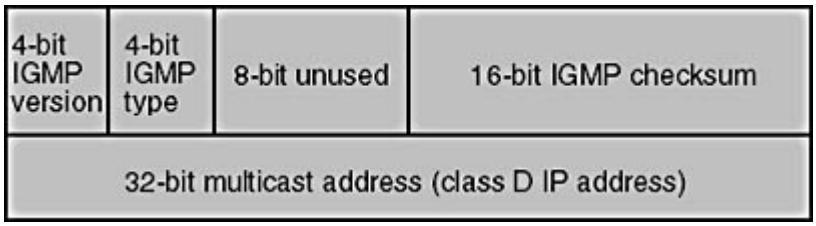

Figure 13-3. *IGMPv1 header format*

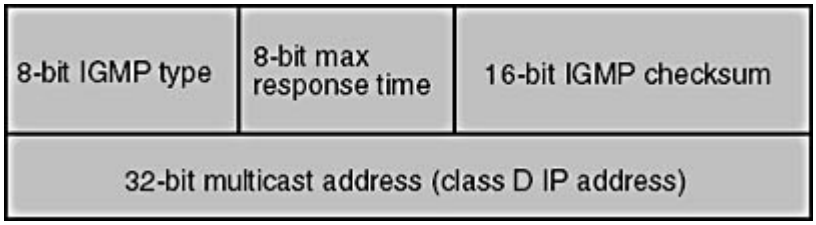

Figure 13-4. *IGMPv2 header format*

Version 1 has two 4-bit fields. The first field is the IGMP version, and the second field is the IGMP message type. In version 2, a single 8-bit field replaces these two fields. Additionally, the unused field in version 1 becomes the maximum response time field. The Max Response Time field is meaningful only in Membership Query messages: it specifies the maximum time allowed before sending a responding report in units of 0.1 second. In all other messages, the sender sets this field to 0 and receivers ignore it.

For IGMPv1, the version field is always set to 1 and the type field is one of two possible values, shown in Table 13-2. Routers use the host membership query (0x1) to determine which multicast groups the host is using. In this case, the group address field is 0. The router sends the packet to the all-hosts address (224.0.0.1). If any hosts are still members of any group, each host sends a host membership report (IGMP message type 0x2) back to the all-routers address with the address to which it is joined. When a host first joins a group, a host membership report message is sent to the all-routers group as well.

As shown in Table 13-3, version 2 of IGMP adds two new message types: version 2 membership report (0x16) and leave group (0x17). Notice that the other two messages are the same as the version 1 messages (host membership query and membership report), even though their assigned values are different. However, remember that the version 2 IGMP packet condenses the version and type fields into a single field, and 0x11 is 00010001 in binary, which neatly makes itself look like an IGMP packet with version equal to 1 and type equal to 1.

Table 13-2. *IGMP version 1 message types*

Type	*Description*
0x1	Host membership query
0x2	Host membership report

Table 13-3. *IGMP version 2 message types*

Type	*Description*
0x11	Membership query
0x12	Version 1 membership report
0x16	Version 2 membership report
0x17	Leave group

The membership query (0x11) in version 2 is different from the membership query in version 1 in one minor way. Not only can it do the general query, as in version 1, it can also query for membership on a network for a specific group address. It does this by sending a membership query packet to the all-hosts group with a specific group in the group address field for which it wants to query. The version 2 membership report is new, as it allows the use of the maximum timeout field that controls the amount of time hosts have to respond to the query before it stops forwarding IP multicast traffic for the given group.

Using *IP_HDRINCL*

The one limitation of raw sockets is that you can work only with certain protocols that are already defined, such as ICMP and IGMP. You cannot create a raw socket with *IPPROTO_UDP* and manipulate the UDP header; likewise with TCP. To manipulate the IP header as well as either the TCP or UDP header (or any other protocol encapsulated in IP), you must use the *IP_HDRINCL* socket option with a raw socket. This option allows you to build your own IP header as well as the headers of other protocols.

Additionally, if you want to implement your own protocol scheme that is encapsulated in IP, you can create a raw socket and use the *IPPROTO_RAW* value as the protocol. This allows you to set the protocol field in the IP header manually and build your own custom protocol header. In this section, we'll take a look at how to build your own UDP packets so that you can gain a good understanding of the steps involved. Once you understand how to manipulate the UDP header, creating your own protocol header or manipulating other protocols encapsulated in IP is fairly trivial.

When you use the *IP_HDRINCL* option, you are required to fill in the IP header yourself for every send call, as well as the headers of any other protocols wrapped within. The IP header is described in Chapter 9 , Figure 9-3, in the section on the *IP_HDRINCL* option. The UDP header is quite a bit simpler than IP. It is only 8 bytes long and contains only four fields, as shown in Figure 13-5. The first two fields are the source and destination port numbers. They are 16 bits each. The third field is the UDP length, which is the length, in bytes, of the UDP header and data. The fourth field is the checksum, which we will discuss shortly. The last part of the UDP packet is the data.

Figure 13-5. *UDP header format*

Because UDP is an unreliable protocol, calculating the checksum is optional; however, we will cover it for the sake of completeness. Unlike the IP checksum, which covers only the IP header, the UDP checksum covers the data and also includes part of the IP header. The additional fields required to calculate the UDP checksum are known as a pseudo-header. A pseudo-header is composed of the following items:

- 32-bit source IP address (IP header)
- 32-bit destination IP address (IP header)
- 8-bit field zeroed out
- 8-bit protocol
- 16-bit UDP length

Added to these items are the UDP header and data. The method of calculating the checksum is the same as for IP and ICMP: the 16-bit one's complement sum. Because the data can be an odd number, it might be necessary to pad a zero byte to the end of the data in order to calculate the checksum. This pad field is not transmitted as part of the data. Figure 13-6 illustrates all the fields required for the checksum calculation. The first three 32-bit words make up the UDP pseudo-header. Following this is the UDP header and its data. Notice that because the checksum is calculated on 16-bit values, the data might need to be padded with a zero byte.

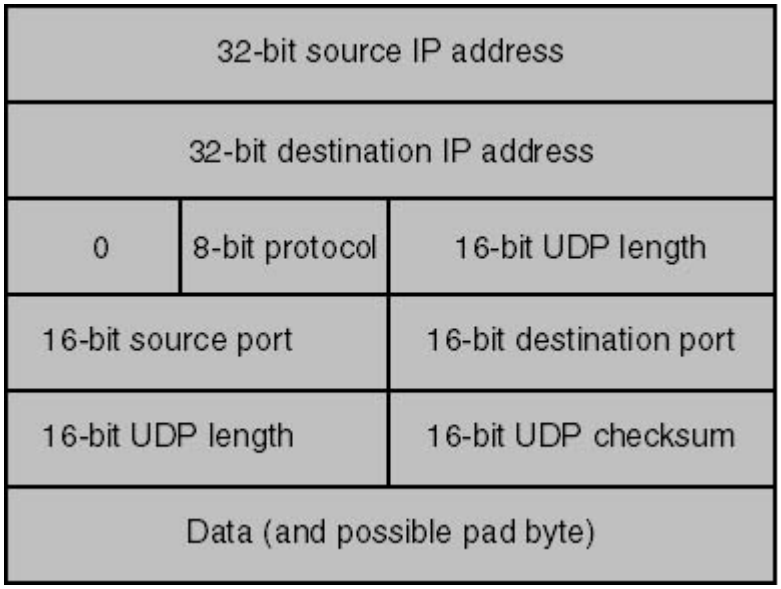

Figure 13-6. *UDP pseudo-header*

Our example program in Figure 13-7 simply sends a UDP packet to any given destination IP and port from any source IP and port of your choice. The first step is to create a raw socket and set the *IP_HDRINCL* flag:

```
SOCKET   s;
BOOL     bOpt;

s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_UDP, NULL, 0,
        WSA_FLAG_OVERLAPPED);
ret = setsockopt(s, IPPROTO_IP, IP_HDRINCL, (char *)&bOpt,
        sizeof(bOpt));
```

Note that we created a raw socket of protocol *IPPROTO_UDP* . This works only on Windows 2000 because it also requires the *IP_HDRINCL* option to be set. The example, Iphdrinc.c, illustrates how to use raw sockets and the *IP_HDRINCL* option to manipulate the IP and UDP headers on outgoing packets.

Figure 13-7 *Raw UDP example*

```
// Module Name: Iphdrinc.c
//
#pragma pack(1)

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <ws2tcpip.h>

#include <stdio.h>
#include <stdlib.h>

#define MAX_MESSAGE        4068
#define MAX_PACKET         4096
//
```

```
// Set up some default values
//
#define DEFAULT_PORT       5150
#define DEFAULT_IP         "10.0.0.1"
#define DEFAULT_COUNT      5
#define DEFAULT_MESSAGE    "This is a test"

//
// Define the IP header. Make the version and length fields one
// character since we can't declare two 4-bit fields without
// the compiler aligning them on at least a 1-byte boundary.
//
typedef struct ip_hdr
{
    unsigned char  ip_verlen;        // IP version & length
    unsigned char  ip_tos;           // IP type of service
    unsigned short ip_totallength;   // Total length
    unsigned short ip_id;            // Unique identifier
    unsigned short ip_offset;        // Fragment offset field
    unsigned char  ip_ttl;           // Time to live
    unsigned char  ip_protocol;      // Protocol(TCP, UDP, etc.)
    unsigned short ip_checksum;      // IP checksum
    unsigned int   ip_srcaddr;       // Source address
    unsigned int   ip_destaddr;      // Destination address
} IP_HDR, *PIP_HDR, FAR* LPIP_HDR;
//
// Define the UDP header
//
typedef struct udp_hdr
{
    unsigned short src_portno;       // Source port number
    unsigned short dst_portno;       // Destination port number
    unsigned short udp_length;       // UDP packet length
    unsigned short udp_checksum;     // UDP checksum (optional)
} UDP_HDR, *PUDP_HDR;

//
// Global variables
//
unsigned long  dwToIP,               // IP to send to
               dwFromIP;             // IP to send from (spoof)
unsigned short iToPort,              // Port to send to
               iFromPort;            // Port to send from (spoof)
DWORD          dwCount;              // Number of times to send
char           strMessage[MAX_MESSAGE]; // Message to send

//
// Description:
//    Print usage information and exit
//
void usage(char *progname)
{
    printf("usage: %s [-fp:int] [-fi:str] [-tp:int] [-ti:str]\
```

```
        [-n:int] [-m:str]\n", progname);
    printf("       -fp:int   From (sender) port number\n");
    printf("       -fi:IP    From (sender) IP address\n");
    printf("       -fp:int   To (recipient) port number\n");
    printf("       -fi:IP    To (recipient) IP address\n");
    printf("       -n:int    Number of times to read message\n");
    printf("       -m:str    Size of buffer to read\n\n");
    ExitProcess(1);
}
//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some global flags to
//    indicate the actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int               i;

    iToPort = DEFAULT_PORT;
    iFromPort = DEFAULT_PORT;
    dwToIP = inet_addr(DEFAULT_IP);
    dwFromIP = inet_addr(DEFAULT_IP);
    dwCount = DEFAULT_COUNT;
    strcpy(strMessage, DEFAULT_MESSAGE);

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'f':        // From address
                    switch (tolower(argv[i][2]))
                    {
                        case 'p':
                            if (strlen(argv[i]) > 4)
                                iFromPort = atoi(&argv[i][4]);
                            break;
                        case 'i':
                            if (strlen(argv[i]) > 4)
                                dwFromIP = inet_addr(&argv[i][4]);
                            break;
                        default:
                            usage(argv[0]);
                            break;
                    }
                    break;
                case 't':        // To address
                    switch (tolower(argv[i][2]))
                    {
                        case 'p':
```

```
                            if (strlen(argv[i]) > 4)
                                iToPort = atoi(&argv[i][4]);
                            break;
                        case 'i':
                            if (strlen(argv[i]) > 4)
                                dwToIP = inet_addr(&argv[i][4]);
                            break;
                        default:
                            usage(argv[0]);
                            break;
                    }
                    break;
                case 'n':        // Number of times to send message
                    if (strlen(argv[i]) > 3)
                        dwCount = atol(&argv[i][3]);
                    break;
                case 'm':
                    if (strlen(argv[i]) > 3)
                        strcpy(strMessage, &argv[i][3]);
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
    }
    return;
}

//
// Function: checksum
//
// Description:
//    This function calculates the 16-bit one's complement sum
//    for the supplied buffer
//
USHORT checksum(USHORT *buffer, int size)
{
    unsigned long cksum=0;

    while (size > 1)
    {
        cksum += *buffer++;
        size  -= sizeof(USHORT);
    }
    if (size)
    {
        cksum += *(UCHAR*)buffer;
    }
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >>16);

    return (USHORT)(~cksum);
```

```
}

//
// Function: main
//
// Description:
//    First parse command line arguments and load Winsock. Then
//    create the raw socket and set the IP_HDRINCL option.
//    Following this, assemble the IP and UDP packet headers by
//    assigning the correct values and calculating the checksums.
//    Then fill in the data and send to its destination.
//
int main(int argc, char **argv)
{
    WSADATA            wsd;
    SOCKET             s;
    BOOL               bOpt;
    struct sockaddr_in remote;       // IP addressing structures
    IP_HDR             ipHdr;
    UDP_HDR            udpHdr;
    int                ret;
    DWORD              i;
    unsigned short     iTotalSize,   // Lots of sizes needed to fill
                       iUdpSize,     // the various headers with
                       iUdpChecksumSize,
                       iIPVersion,
                       iIPSize,
                       cksum = 0;
    char               buf[MAX_PACKET],
                      *ptr = NULL;
    IN_ADDR            addr;

    // Parse command line arguments, and print them out
    //
    ValidateArgs(argc, argv);
    addr.S_un.S_addr = dwFromIP;
    printf("From IP: <%s>\n     Port: %d\n", inet_ntoa(addr),
        iFromPort);
    addr.S_un.S_addr = dwToIP;
    printf("To   IP: <%s>\n     Port: %d\n", inet_ntoa(addr),
        iToPort);
    printf("Message: [%s]\n", strMessage);
    printf("Count:   %d\n", dwCount);

    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("WSAStartup() failed: %d\n", GetLastError());
        return -1;
    }
    //  Creating a raw socket
    //
    s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_UDP, NULL, 0,0);
    if (s == INVALID_SOCKET)
```

```
{
    printf("WSASocket() failed: %d\n", WSAGetLastError());
    return -1;
}

// Enable the IP header include option
//
bOpt = TRUE;
ret = setsockopt(s, IPPROTO_IP, IP_HDRINCL, (char *)&bOpt,
   sizeof(bOpt));
if (ret == SOCKET_ERROR)
{
    printf("setsockopt(IP_HDRINCL) failed: %d\n", WSAGetLastError());
    return -1;
}
// Initalize the IP header
//
iTotalSize = sizeof(ipHdr) + sizeof(udpHdr) + strlen(strMessage);

iIPVersion = 4;
iIPSize = sizeof(ipHdr) / sizeof(unsigned long);
//
// IP version goes in the high-order 4 bits of ip_verlen. The
// IP header length (in 32-bit words) goes in the lower 4 bits.
//
ipHdr.ip_verlen = (iIPVersion << 4) | iIPSize;
ipHdr.ip_tos = 0;                          // IP type of service
ipHdr.ip_totallength = htons(iTotalSize); // Total packet len
ipHdr.ip_id = 0;                 // Unique identifier: set to 0
ipHdr.ip_offset = 0;             // Fragment offset field
ipHdr.ip_ttl = 128;              // Time to live
ipHdr.ip_protocol = 0x11;        // Protocol(UDP)
ipHdr.ip_checksum = 0 ;          // IP checksum
ipHdr.ip_srcaddr = dwFromIP;     // Source address
ipHdr.ip_destaddr = dwToIP;      // Destination address
//
// Initalize the UDP header
//
iUdpSize = sizeof(udpHdr) + strlen(strMessage);

udpHdr.src_portno = htons(iFromPort) ;
udpHdr.dst_portno = htons(iToPort) ;
udpHdr.udp_length = htons(iUdpSize) ;
udpHdr.udp_checksum = 0 ;
//
// Build the UDP pseudo-header for calculating the UDP checksum.
// The pseudo-header consists of the 32-bit source IP address,
// the 32-bit destination IP address, a zero byte, the 8-bit
// IP protocol field, the 16-bit UDP length, and the UDP
// header itself along with its data (padded with a 0 if
// the data is odd length).
//
iUdpChecksumSize = 0;
```

```
ptr = buf;
ZeroMemory(buf, MAX_PACKET);

memcpy(ptr, &ipHdr.ip_srcaddr,  sizeof(ipHdr.ip_srcaddr));
ptr += sizeof(ipHdr.ip_srcaddr);
iUdpChecksumSize += sizeof(ipHdr.ip_srcaddr);

memcpy(ptr, &ipHdr.ip_destaddr, sizeof(ipHdr.ip_destaddr));
ptr += sizeof(ipHdr.ip_destaddr);
iUdpChecksumSize += sizeof(ipHdr.ip_destaddr);

ptr++;
iUdpChecksumSize += 1;

memcpy(ptr, &ipHdr.ip_protocol, sizeof(ipHdr.ip_protocol));
ptr += sizeof(ipHdr.ip_protocol);
iUdpChecksumSize += sizeof(ipHdr.ip_protocol);

memcpy(ptr, &udpHdr.udp_length, sizeof(udpHdr.udp_length));
ptr += sizeof(udpHdr.udp_length);
iUdpChecksumSize += sizeof(udpHdr.udp_length);

memcpy(ptr, &udpHdr, sizeof(udpHdr));
ptr += sizeof(udpHdr);
iUdpChecksumSize += sizeof(udpHdr);

for(i = 0; i < strlen(strMessage); i++, ptr++)
    *ptr = strMessage[i];
iUdpChecksumSize += strlen(strMessage);

cksum = checksum((USHORT *)buf, iUdpChecksumSize);
udpHdr.udp_checksum = cksum;
//
// Now assemble the IP and UDP headers along with the data
// so we can send it
//
ZeroMemory(buf, MAX_PACKET);
ptr = buf;

memcpy(ptr, &ipHdr, sizeof(ipHdr));   ptr += sizeof(ipHdr);
memcpy(ptr, &udpHdr, sizeof(udpHdr)); ptr += sizeof(udpHdr);
memcpy(ptr, strMessage, strlen(strMessage));

// Apparently, this SOCKADDR_IN structure makes no difference.
// Whatever we put as the destination IP addr in the IP header
// is what goes. Specifying a different destination in remote
// will be ignored.
//
remote.sin_family = AF_INET;
remote.sin_port = htons(iToPort);
remote.sin_addr.s_addr = dwToIP;

for(i = 0; i < dwCount; i++)
```

```
    {
        ret = sendto(s, buf, iTotalSize, 0, (SOCKADDR *)&remote,
            sizeof(remote));
        if (ret == SOCKET_ERROR)
        {
            printf("sendto() failed: %d\n", WSAGetLastError());
            break;
        }
        else
            printf("sent %d bytes\n", ret);
    }
    closesocket(s) ;
    WSACleanup() ;

    return 0;
}
```

After creating the socket and setting the *IP_HDRINCL* option, the code begins filling in the IP header. You'll notice that the code declared the IP header as a structure: *IP_HDR* . Notice that the first two 4-bit fields have been combined into a single field because the compiler can align the fields only on a minimum of a 1-byte boundary. Because of this, the code has to manipulate the IP version into the high 4 bits. The *ip_protocol* field is set to 0x11, which corresponds to UDP. The code also sets the *ip_srcaddr* field to the source IP address (or whatever address you want the recipient to think it came from) and the *ip_destaddr* field to the recipient's IP address. The network stack calculates the IP checksum, so the code doesn't have to set it.

The next step is to initialize the UDP header. This is simple because there aren't as many fields. The source and destination port numbers are set along with the UDP header size. The checksum field is initialized to 0. Although your code is not required to calculate the UDP checksum, this example does in order to illustrate how the calculation is done with the UDP pseudo-header. To make the calculation easier, we copy all the necessary fields into a temporary character buffer, *buf* . Then we pass this buffer to our checksum calculating function along with the length of the buffer to perform the calculation over.

The last thing to do before sending the datagram is to assemble the various pieces of the message in one contiguous buffer. Simply use the *memcpy* function to copy the IP and UDP headers followed by the data into a contiguous buffer. Next call the *sendto* function to send the data. Note that when you use the *IP_HDRINCL* option, the *to* parameter in *sendto* is ignored. The data is always sent to the host that you specify in the IP header.

To see Iphdrinc.c in action, you can use the UDP receiver program from Chapter 7 , Receiver.c. For example, start the UDP receiver application with the following parameters:

```
Receiver.exe -p:5150 -i:xxx.xxx.xxx.xxx -n:5 -b:1000
```

You can leave off the *-i* parameter if the machine has only one network interface; otherwise, specify the IP address of one of the interfaces. On the Windows 2000 machine, start the Iphdrinc.exe example as

```
Iphdrinc.exe -fi:1.2.3.4 -fp:10 -ti:xxx.xxx.xxx.xxx -tp:5150 -n:5150
```

The IP address given for the -ti parameter should be the same IP address that the receiver listens on. The receiver application should report the following output:

```
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
```

```
[1.2.3.4:10] sent me: 'This is a test'
[1.2.3.4:10] sent me: 'This is a test'
```

After the call to *recvfrom* completes, Receiver.c prints the message along with the address information returned from the *SOCKADDR_IN* structure passed into *recvfrom* . You can further verify that the IP header for the packet transmitted on the wire is the same as the one you created by using the Microsoft Network Monitor application to capture the packets on the network. Table 13-4 lists the possible command line parameters for the Iphdrinc example. You can specify the source and destination IP address and port number.

Table 13-4. *Iphdrinc.c parameters*

Parameter	*Description*
-fi:*xxx.xxx.xxx.xxx*	Source IP address of the IP packet
-fp:*int*	Source port number of the IP packet
-ti:*xxx.xxx.xxx.xxx*	Destination IP address of the IP packet
-tp:*int*	Destination port of the IP packet
-n:*int*	Number of UDP datagrams to send
-m:*string*	Message to send

Conclusion

Raw sockets are a powerful mechanism to manipulate the underlying protocol itself. This chapter illustrated how raw sockets can be used to create ICMP and IGMP applications through Winsock, but raw sockets can be used in a multitude of other applications—too many to discuss in a single chapter. To take full advantage of the capabilities of raw sockets and the *IP_HDRINCL* option, you must thoroughly understand the IP protocol as well as any protocols encapsulated in IP.

Chapter 14

Winsock 2 Service Provider Interface

The Winsock 2 Service Provider Interface (SPI) represents the other side of programming for Winsock (as compared with the Winsock 2 API). On one side of Winsock you have an API, and on the other you have an SPI. Chapters 6 through 13 explained the Winsock 2 API. Winsock 2 is designed around the Windows Open System Architecture (WOSA), which has a standardized API between Winsock and Winsock applications and a standardized SPI between Winsock and Winsock service providers (such as TCP/IP). Figure 14-1 on the following page shows how Ws2_32.dll, the Winsock 2 support dynamic-link library (DLL), is layered between Winsock applications and Winsock service providers. This chapter explains the Winsock 2 SPI in detail. When you finish this chapter, you will understand how to extend the capabilities of Winsock 2 by developing a service provider.

SPI Basics

The Winsock 2 SPI allows you to develop two types of service providers—transport providers and name space providers. *Transport providers* (commonly referred to as protocol stacks, such as TCP/IP) are services that supply functions that set up communication, transfer data, exercise data-flow control and error control, and so on. *Name space providers* are services that associate the addressing attributes of a network protocol with one or more user-friendly names and thus enable protocol-independent name resolution. Service providers are nothing more than Win32 support DLLs that are hooked below Winsock 2's Ws2_32.dll module. They provide the inner workings of many of the calls defined in the Winsock 2 API.

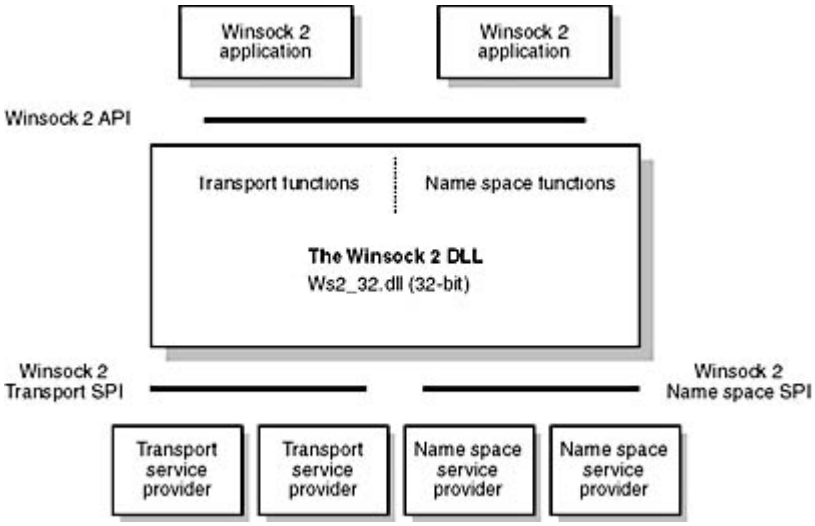

Figure 14-1. *Winsock 2's WOSA-compliant architecture*

SPI Naming Conventions

The Winsock 2 SPI function prototypes use the following function prefix naming conventions:

- *WSP* (WinSock Provider) Identifies transport service provider functions
- *NSP* (name space provider) Identifies name space provider functions
- *WPU* (Winsock provider upcall) Identifies Ws2_32.dll support functions called by service providers
- *WSC* (Winsock configuration) Identifies functions for installing service providers in Winsock 2

For example, a function named *WSCInstallProvider* is simply an SPI configuration function.

Mapping Between Winsock 2 API and SPI Functions

In most cases, when an application calls a Winsock 2 function, Ws2_32.dll calls a corresponding Winsock 2 SPI function to carry out the requested functionality using a specific service provider. For example, *select* maps to *WSPSelect*, *WSAConnect* maps to *WSPConnect*, and *WSAAccept* maps to *WSPAccept*. However, not all Winsock functions have a corresponding SPI function. The following list details these exceptions.

- Support functions such as *htonl*, *htons*, *ntohl*, and *ntohs* are implemented within Ws2_32.dll and aren't passed down to a service provider. The same holds true for the WSA versions of these functions.
- IP conversion functions such as *inet_addr* and *inet_ntoa* are implemented only within Ws2_32.dll.

- All of the IP-specific name conversion and resolution functions in Winsock 1.1 such as *getXbyY, WSAAsyncGetXByY,* and *WSACancelAsyncRequest*, as well as *gethostname,* are implemented within Ws2_32.dll.

- Winsock service provider enumeration and the blocking hook_related functions are implemented within Ws2_32.dll. Thus *WSAEnumProtocols*, *WSAIsBlocking*, *WSASetBlockingHook*, and *WSAUnhookBlockingHook* do not appear as SPI functions.

- Winsock error codes are managed within Ws2_32.dll. *WSAGetLastError* and *WSASetLastError* aren't needed in the SPI.

- The event object manipulation and wait functions—including *WSACreateEvent*, *WSACloseEvent*, *WSASetEvent*, *WSAResetEvent*, and *WSAWaitForMultipleEvents*—are mapped directly to native Win32 operating system calls and aren't present in the SPI.

Now you're ready to learn which Winsock APIs are mapped to Winsock 2 service providers. When you develop a service provider, you will find all SPI function prototypes defined in the include file Ws2spi.h.

Transport Service Providers

Two types of transport service providers are used in Winsock 2: base service providers and layered service providers. *Base* service providers implement the actual details of a network transport protocol—such as TCP/IP—including core network protocol functions such as sending and receiving data on a network. *Layered* service providers implement only higher-level custom communication functions and rely on an underlying base service provider for the actual exchange of data on a network. For example, you can implement a data security manager or a bandwidth manager on top of an existing base TCP/IP provider. Figure 14-2 shows how one or more layered providers can be installed between Ws2_32.dll and a base provider.

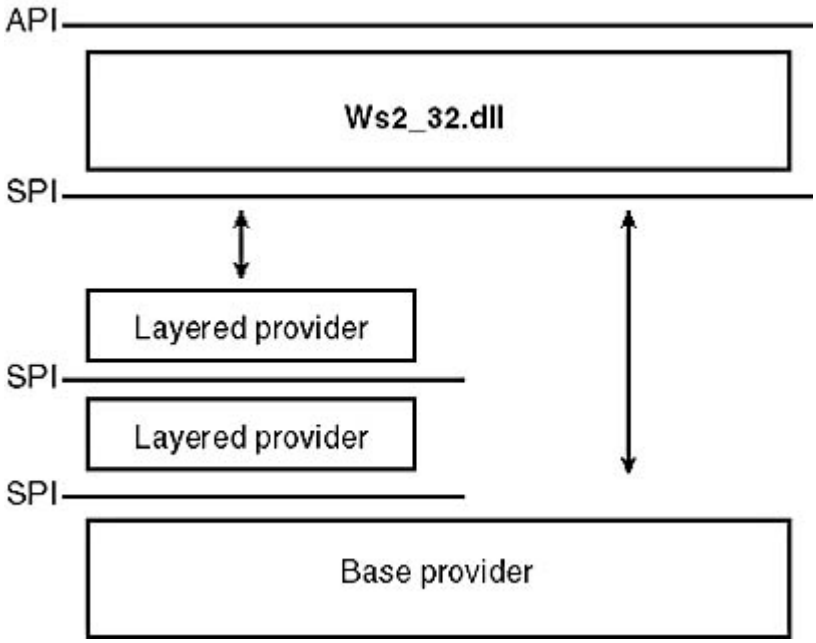

Figure 14-2. *Layered provider architecture*

This section focuses on the development aspects of a layered transport service provider. If you're developing a base service provider, the principles described here apply. However, we don't detail what is involved in implementing a particular SPI function from the ground up. For example, we don't supply the details of how the *WSPSend* SPI function writes data to a network adapter. Instead we show how the *WSPSend* function of a layered provider invokes the *WSPSend* function of a lower provider, which is a requirement of most layered service providers. Essentially, most of the work involved in developing a layered provider is relaying the SPI calls of your provider to the next provider below you. The tricky part is handling I/O calls from the Winsock I/O models described in Chapter 8, which we will discuss later in this chapter. On the companion CD-ROM, we provide an example named LSP that demonstrates how to implement a layered service provider that simply counts how many bytes are transmitted over a socket using the IP transport protocol. The Microsoft Platform SDK also features a more advanced layered service provider example named "layered" that can be found in the MSDN Platform SDK examples downloaded from *ftp://ftp.microsoft.com/bussys/WinSock/winsock2/layered.zip.*

NOTE

Throughout this section on transport service providers, we often use the terms "SPI client" and "lower provider." An *SPI client* can be either Winsock 2's Ws2_32.dll or another layered service provider placed above your service provider. An SPI client is never a Winsock application itself because Winsock applications must use the Winsock 2 API exported from Ws2_32.dll. The term "lower provider" is used only when we describe development aspects of a layered service provider. A lower provider can be either another layered service provider or a base server provider. As you will see, you can have multiple layered service providers installed on a machine; thus there is a chance that a layered provider might be installed below your provider.

WSPStartup

Winsock 2 transport service providers are implemented as standard Windows dynamic-link library modules in which you must export a *DllMain* function. Additionally, you must export a single function entry named *WSPStartup*. When a caller (the SPI client) invokes *WSPStartup*, it exposes 30 additional SPI functions that make up a transport service provider via a function dispatch table passed as a parameter. (See Table 14-1.) Your service provider must provide an implementation of *WSPStartup* plus all 30 functions.

Table 14-1. *Transport provider support functions*

API Function	*Maps to SPI Function*
WSAAccept (accept also indirectly maps to WSPAccept)	*WSPAccept*
WSAAddressToString	*WSPAddressToString*
WSAAsyncSelect	*WSPAsyncSelect*
bind	*WSPBind*
WSACancelBlockingCall	*WSPCancelBlockingCall*
WSACleanup	*WSPCleanup*
closesocket	*WSPCloseSocket*
WSAConnect (connect also indirectly maps to WSPConnect)	*WSPConnect*
WSADuplicateSocket	*WSPDuplicateSocket*
WSAEnumNetworkEvents	*WSPEnumNetworkEvents*
WSAEventSelect	*WSPEventSelect*
WSAGetOverlappedResult	*WSPGetOverlappedResult*
getpeername	*WSPGetPeerName*
getsockname	*WSPGetSockName*
getsockopt	*WSPGetSockOpt*
WSAGetQOSByName	*WSPGetQOSByName*
WSAIoctl	*WSPIoctl*
WSAJoinLeaf	*WSPJoinLeaf*
listen	*WSPListen*
WSARecv (recv also indirectly maps to WSPRecv)	*WSPRecv*
WSARecvDisconnect	*WSPRecvDisconnect*
WSARecvFrom (recvfrom also indirectly maps to WSPRecvFrom)	*WSPRecvFrom*
select	*WSPSelect*
WSASend (send also indirectly maps to WSPSend)	*WSPSend*
WSASendDisconnect	*WSPSendDisconnect*
WSASendTo (sendto also indirectly maps to WSPSendTo)	*WSPSendTo*
setsockopt	*WSPSetSockOpt*
shutdown	*WSPShutdown*

WSASocket (socket also indirectly maps to WSPSocket)	*WSPSocket*
WSAStringToAddress	*WSPStringToAddress*

It is important to understand how and when *WSPStartup* is called. You might be compelled to think it is called when an application calls the *WSAStartup* API. This is not the case. Winsock does not know the type of service provider it needs to use during *WSAStartup*. Winsock determines which service provider it needs to load based on the address family, socket type, and protocol parameters of a *WSASocket* call. Therefore, Winsock invokes a service provider only when an application creates a socket through the *socket* or *WSASocket* API call. For example, if an application creates a socket using the address family *AF_INET* and the socket type *SOCK_STREAM*, Winsock searches and loads an appropriate transport provider that provides TCP/IP functionality. We will describe this loading process in more detail in this chapter's section on installing transport service providers.

Parameters

WSPStartup is the key function used to initialize the functionality of a transport service provider and is defined as

```
int WSPStartup(
    WORD wVersionRequested,
    LPWSPDATAW lpWSPData,
    LPWSAPROTOCOL_INFOW lpProtocolInfo,
    WSPUPCALLTABLE UpcallTable,
    LPWSPPROC_TABLE lpProcTable
);
```

The *wVersionRequested* parameter receives the latest version of Windows Sockets SPI support that the caller can use. Your service provider should check this value to see whether it can support the requested version. Your provider uses the *lpWSPData* parameter to return version information about itself through a *WSPDATA* structure, which is defined as

```
typedef struct WSPData
{
    WORD         wVersion;
    WORD         wHighVersion;
    WCHAR        szDescription[WSPDESCRIPTION_LEN + 1];
} WSPDATA, FAR * LPWSPDATA;
```

In the *wVersion* field, your provider must return which version of Winsock the caller is expected to use. The *wHighVersion* parameter must return the highest Winsock version supported by your provider. This is usually the same value as *wVersion*. (Winsock versioning information is described in detail in Chapter 7.) The *szDescription* field returns a null-terminated UNICODE string identifying your provider to an SPI client. This field can contain up to 256 characters.

The *lpProtocolInfo* parameter of *WSPStartup* is a pointer to a *WSAPROTOCOL_ INFOW* structure that contains characteristic information about your provider. (Protocol characteristics and the details of this structure are described in "Winsock 2 Protocol Information" in Chapter 5.) The information in *WSAPROTOCOL_INFOW* is retrieved by Ws2_32.dll from the Winsock 2 service provider catalog that contains property information about service providers. We will further describe Winsock 2 catalog entries in the section on installing transport service providers.

When you develop a layered service provider, you need to treat the *lpProtocolInfo* parameter in a unique way because it contains information on how your provider is layered between Ws2_32.dll and a base service provider. This parameter is used to determine the next service provider below your provider. (It could be another layered provider or possibly a base provider.) At some point, your provider must load the next service provider by loading

the next provider's DLL module and calling the provider's *WSPStartup* function. The *WSAPROTOCOL_INFOW* structure pointed to by *lpProtocolInfo* contains a field, *ProtocolChain*, that identifies how your service provider is ordered with others on a machine.

The *ProtocolChain* field is actually a *WSAPROTOCOLCHAIN* structure that is defined as

```
typedef struct _WSAPROTOCOLCHAIN
{
    int ChainLen;
    DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN, FAR * LPWSAPROTOCOLCHAIN;
```

The *ChainLen* field identifies how many layers are sandwiched between Ws2_32.dll and a base service provider. (This number also includes the base provider.) If you have a computer that has only one layered service provider above a protocol, such as TCP/IP, this value will be 2. The *ChainEntries* field is an array of service-provider catalog identification numbers that uniquely identify the layered service providers that are linked together for a particular protocol. We will describe the *WSAPROTOCOLCHAIN* structure in the section on installing transport service providers later in this chapter. One requirement of a layered service provider is to search the *ProtocolChain* field of the structure to determine its own location in the array of service providers (by searching for your layer's own catalog entry) and also to determine the next provider in the array. If the next provider is another layer, you must pass the *lpProtocolInfo* structure unmodified to the next layer's *WSPStartup* function. If the next layer is the last element in the array (indicating a base provider), your provider must use the base provider's *WSAPROTOCOL_INFOW* structure to perform a substitution on the *lpProtocolInfo* structure when calling the base provider's *WSPStartup* function. Figure 14-3 demonstrates how a layered provider should programmatically manage the *lpProtocolInfo* structure.

Figure 14-3. *Finding the appropriate WSAPROTOCOL_INFOW structure for WSPStartup*

```
LPWSAPROTOCOL_INFOW ProtocolInfo;
LPWSAPROTOCOL_INFOW ProtoInfo = lpProtocolInfo;
DWORD ProtocolInfoSize = 0;

// Find out how many entries we need to enumerate
if (WSCEnumProtocols(NULL, ProtocolInfo, &ProtocolInfoSize,
    &ErrorCode) == SOCKET_ERROR)
{
    if (ErrorCode != WSAENOBUFS)
    {
        return  WSAEPROVIDERFAILEDINIT;
    }
}

if ((ProtocolInfo = (LPWSAPROTOCOL_INFOW) GlobalAlloc(GPTR,
    ProtocolInfoSize)) == NULL)
{
    return  WSAEPROVIDERFAILEDINIT;
}

if ((TotalProtocols = WSCEnumProtocols(NULL, ProtocolInfo,
    &ProtocolInfoSize, &ErrorCode)) == SOCKET_ERROR)
{
    return  WSAEPROVIDERFAILEDINIT;
}
```

```
// Find our layered provider's catalog ID entry
for (i = 0; i < TotalProtocols; i++)
    if (memcmp (&ProtocolInfo[i].ProviderId, &ProviderGuid,
        sizeof (GUID))==0)
    {
        gLayerCatId = ProtocolInfo[i].dwCatalogEntryId;
        break;
    }

// Save our provider's catalog ID entry
gChainId = lpProtocolInfo->dwCatalogEntryId;

// Find our catalog ID entry in the protocol chain
for(j = 0; j < lpProtocolInfo->ProtocolChain.ChainLen; j++)
{
    if (lpProtocolInfo->ProtocolChain.ChainEntries[j] ==
        gLayerCatId)
    {
        NextProviderCatId =
        lpProtocolInfo->ProtocolChain.ChainEntries[j+1];

        // Check whether next provider is the base provider
        if (lpProtocolInfo->ProtocolChain.ChainLen ==
            (j + 2))
        {
            for (i = 0; i < TotalProtocols; i++)
                if (NextProviderCatId ==
                    ProtocolInfo[i].dwCatalogEntryId)
                {
                    ProtoInfo = &ProtocolInfo[i];
                    break;
                }
        }
        break;
    }
}

// At this point, ProtoInfo will contain the appropriate
// WSAPROTOCOL_INFOW structure
```

The *UpcallTable* parameter of *WSPStartup* receives Ws2_32.dll's SPI upcall dispatch table, which contains pointers to support functions that your provider can use to manage I/O operations between itself and Winsock 2. We will define most of these functions and describe how to use them in this chapter's section on Winsock I/O model support.

The final parameter of *WSPStartup*—*lpProcTable*—represents a table of 30 SPI function pointers that your service provider must implement. These functions are listed in Table 14-1. Each SPI function conforms to the parameter specification of its API counterpart with the following two adjustments:

- Each function supplies a final parameter, *lpErrno*, which your provider must use to report specific Winsock error code information when your implementation fails. For example, if you're implementing *WSPSend* and you cannot allocate memory, you might want to return the Winsock error *WSAENOBUFS*.
- The SPI functions *WSPSend*, *WSPSendTo*, *WSPRecv*, *WSPRecvFrom*, and *WSPIoctl* feature an additional

parameter, *lpThreadId*, which identifies the application thread that invoked the SPI function. As we will see later in this chapter, this feature is useful for supporting completion routines.

A final item worth noting is that several Winsock 1.1 functions, such as *send* and *recv*, map directly to their corresponding Winsock 2 functions. We claim that these Winsock 1.1 functions indirectly map to the corresponding SPI function because they actually call a Winsock 2 function featuring similar functionality. For example, the *send* function actually calls the *WSASend* function, which maps to *WSPSend*. In Table 14-1, we noted the functions that indirectly map to SPI functions.

Instance count

In the Winsock specification, applications can call the *WSAStartup* and *WSACleanup* functions an unlimited number of times. Your service provider's *WSPStartup* and *WSPCleanup* functions will be called as many times as their API equivalents. As a result, your service provider should maintain an instance count for how many times the *WSPStartup* call is invoked, and you should decrement this count for each *WSPCleanup*. The purpose of maintaining an instance count is to allow you to streamline the initialization and cleanup procedures of your service provider. For example, if your provider allocates memory to manage internal structures, you can hold on to this memory as long as your instance count is greater than 0. In addition, when your instance count falls to 0, Ws2_32.dll will eventually unload your provider from memory.

Socket Handles

A service provider must return socket handles when an SPI client calls the *WSPSocket*, *WSPAccept*, and *WSPJoinLeaf* functions. Socket handles returned to SPI clients can be either Win32 installable file system (IFS) handles or non-IFS handles. If a service provider returns IFS handles, it is known as an IFS provider; otherwise, it's a non-IFS provider. All of the Microsoft base transport providers are IFS providers.

Winsock is designed to allow Winsock applications to use the Win32 API functions *ReadFile* and *WriteFile* to receive and send data on a socket handle. Therefore, you have to consider how socket handles are created in a service provider. If you develop a service provider with the intention of providing service to Winsock applications that call *ReadFile* and *WriteFile* on a socket handle, consider developing an IFS provider. However, be aware of I/O limitations if you take this approach.

IFS provider

As we mentioned earlier, transport service providers can be layered or base. If you're developing a base IFS provider, your provider will have a kernel-mode operating system component, and this component enables the Winsock provider to create handles that can be used like file-system handles in *ReadFile* and *WriteFile*. Kernel-mode software development is beyond the scope of this book. If you want to know more about how to develop kernel-mode operating system components for Windows, consult the MSDN Device Development Kit (DDK) for more information.

A layered service provider can also become an IFS provider, but only if the layered provider is layered on top of an existing base IFS provider. This involves passing a lower IFS provider's socket handles—retrieved in your layered provider—directly up to your SPI client. Passing socket handles directly up from a lower provider limits a layered provider's functionality in the following ways:

- The layered provider's *WSPSend* and *WSPRecv* functions will not be called if the *ReadFile* and *WriteFile* functions are called on a socket. These functions will bypass a layered provider and directly invoke the base IFS provider's implementation.
- A layered provider will not be able to postprocess overlapped I/O requests that are submitted to a completion port. Postprocessing of the completion port completely bypasses a layered provider.

If your layered provider intends to monitor all I/O passed through the provider, you will have to develop a non-IFS layered provider, which we will discuss later in this chapter.

Whenever an IFS provider (either layered or base) creates a new socket descriptor, the provider is required to call *WPUModifyIFSHandle* prior to supplying the new handle to an SPI client. This allows the Winsock Ws2_32.dll to greatly streamline the process of identifying the IFS service provider associated with a given socket when Win32 APIs such as *ReadFile* and *WriteFile* perform I/O on a socket. *WPUModifyIFSHandle* is defined as

```
SOCKET WPUModifyIFSHandle(
    DWORD dwCatalogEntryId,
    SOCKET ProposedHandle,
    LPINT lpErrno
);
```

The *dwCatalogEntryId* field identifies the catalog ID of your service provider. The *ProposedHandle* parameter represents an IFS handle allocated by your service provider (in the case of a base provider). If you're developing a layered IFS provider, this handle will be passed up from a lower provider. The *lpErrno* parameter receives specific Winsock error code information if this function fails with the return value *INVALID_SOCKET*.

Non-IFS provider

If you're developing a layered provider and intend to monitor every read and write operation that occurs on a socket, you will have to develop a non-IFS provider. Non-IFS providers create socket handles by using the upcall *WPUCreateSocketHandle*. Socket handles created by *WPUCreateSocketHandle* are similar to IFS provider handles in that they also allow Winsock applications to use *ReadFile* and *WriteFile* functions on a socket. However, there is a significant I/O performance penalty associated with these two functions because the Winsock 2 architecture has to perform I/O redirection to a service provider's *WSPRecv* and *WSPSend* functions, respectively. *WPUCreateSocketHandle* is defined as

```
SOCKET WPUCreateSocketHandle(
    DWORD dwCatalogEntryId,
    DWORD dwContext,
    LPINT lpErrno
);
```

The *dwCatalogEntryId* identifies the catalog ID of your service provider. The *dwContext* parameter allows you to associate provider data with a socket descriptor. Note that *dwContext* gives you a lot of freedom in terms of the information you can associate with a socket descriptor. In the LSP sample on the companion CD-ROM, we take advantage of this field by storing the byte count of send and receive data. Winsock provides an upcall *WPUQuerySocketHandleContext* that can be used to retrieve associated socket provider data saved in *dwContext*, and this upcall is defined as

```
int WPUQuerySocketHandleContext(
    SOCKET s,
    LPDWORD lpContext,
    LPINT lpErrno
);
```

The *s* parameter represents the socket handle passed down from an SPI client (originally created from *WPUCreateSocketHandle*) that you want to retrieve socket provider data. The *lpContext* parameter receives the provider data originally passed in *WPUCreateSocketHandle*. The *lpErrno* parameter in both functions receives a specific Winsock error code if these functions fail. If *WPUCreateSocketHandle* fails, it returns *INVALID_SOCKET*; and if *WPUQuerySocketHandleContext* fails, it returns *SOCKET_ERROR*.

Winsock I/O Model Support

As you learned in Chapter 8, Winsock features several I/O models that can be used to manage I/O on a socket. From a service provider's point of view, each model requires using some of the SPI upcall functions provided by Ws2_32.dll that are available from the *UpcallTable* parameter in *WSPStartup,* mentioned earlier. If you're developing a simple IFS layered provider, the principles of each I/O model described in the following sections don't apply—you simply rely on the lower provider to manage all of the I/O for each model. The principles outlined in these sections do apply to any other type of provider. We'll focus our discussion on development aspects of a non-IFS layered service provider.

Blocking and nonblocking

Blocking I/O is the simplest form of I/O in Winsock 2. Any I/O operation with a blocking socket won't return until the operation has completed. Therefore, any thread can execute only one I/O operation at a time. For example, when an SPI client calls the *WSPRecv* function in a blocking fashion, your provider only needs to relay the call directly to the next provider's *WSPRecv* call. Your provider's *WSPRecv* function will return only when the next provider's *WSPRecv* call completes.

Even though blocking I/O is easy to implement, you still have to consider backward compatibility with Winsock 1.1 blocking hooks. The *WSASetBlockingCall* and *WSACancelBlocking* API calls have been removed from the Winsock 2 API specification. However, *WSPCancelBlockingHook* can still be called by Ws2_32.dll if a Winsock 1.1 application calls the *WSASetBlockingHook* and *WSACancelBlockingCall* functions. In a layered service provider, you can simply relay the *WSPCancelBlockingHook* call to the base provider's call. If you're implementing a base provider and a blocking call is in progress, you must implement a mechanism to call the *WPUQueryBlockingCallback* function periodically. *WPUQueryBlockingCallback* is defined as

```
int WPUQueryBlockingCallback(
    DWORD dwCatalogEntryId,
    LPBLOCKINGCALLBACK FAR *lplpfnCallback,
    LPDWORD lpdwContext,
    LPINT lpErrno
);
```

The *dwCatalogEntryId* parameter receives the catalog entry ID of your provider as described in the *WSPStartup* function. The *lplpfnCallback* parameter is a function pointer to the application's blocking hook function that you must call periodically to prevent the application's blocking calls from truly blocking. The callback function has the following form:

```
typedef BOOL (CALLBACK FAR * LPBLOCKINGCALLBACK)(
    DWORD dwContext
);
```

When your provider periodically invokes *lplpfnCallback*, pass the value of the *lpdwContext* parameter to the callback function's *dwContext* parameter. The final parameter of *WPUQueryBlockingCallback*, *lpErrno*, returns Winsock error code information if this function returns *SOCKET_ERROR.*

select

The *select* I/O model requires that a provider manage the *fd_set* structures for the parameters *readfds*, *writefds*, and *exceptfds* in the *WSPSelect* function, which is defined as

```
int WSPSelect(
    int nfds,
    fd_set FAR *readfds,
    fd_set FAR *writefds,
    fd_set FAR *exceptfds,
    const struct timeval FAR *timeout,
    LPINT lpErrno
);
```

Essentially, the *fd_set* data type represents a collection of sockets. When an SPI client calls your provider using *WSPSelect*, it passes socket handles to one or more of these sets. Your provider is responsible for determining when network activities have occurred on each of the sockets listed.

For a non-IFS layered provider, this requires creating three *fd_set* data fields and mapping the SPI client's socket handles to the lower provider's socket handles in each set. Once all the sets are set up, your provider calls *WSPSelect* on the lower provider. When the lower provider completes, your provider has to determine which sockets have events pending in each of the *fd_set* fields. There is a useful upcall *WPUFDIsSet* that determines which lower provider sockets are set. This upcall is similar to the *FD_ISSET* macro discussed in Chapter 8 and is defined as

```
int WPUFDIsSet(
    SOCKET s,
    FD_SET FAR *set
);
```

The *s* parameter represents the socket you're looking for in a set. The *set* parameter is an actual set of socket descriptors. Since your provider will check the contents of each set passed to the lower provider, your provider should save the socket mappings from the upper provider to the lower provider. When the lower provider completes the *WSPSelect* call, you can easily map back which sockets have I/O pending for the upper provider.

Once you have determined which lower provider sockets have network events pending, you have to update the originating *fd_set* sets that were passed in from the originating SPI client. The Ws2spi.h file provides the three macros—*FD_CLR*, *FD_SET*, and *FD_ZERO*—that can be used to manage the originating sets. These macros are described in Chapter 8. Figure 14-4 demonstrates one possibility for implementing *WSPSelect*.

Figure 14-4. *WSPSelect implementation details*

```
int WSPAPI WSPSelect(
    int nfds,
    fd_set FAR * readfds,
    fd_set FAR * writefds,
    fd_set FAR * exceptfds,
    const struct timeval FAR * timeout,
    LPINT lpErrno)
{
    SOCK_INFO *SocketContext;
    u_int i;
    u_int count;
    int Ret;
    int HandleCount;
```

```
    // Build an Upper and Lower provider socket mapping table
    struct
    {
        SOCKET ClientSocket;
        SOCKET ProvSocket;
    } Read[FD_SETSIZE], Write[FD_SETSIZE], Except[FD_SETSIZE];

    fd_set ReadFds, WriteFds, ExceptFds;

    // Build the ReadFds set for the lower provider
    if (readfds)
    {
        FD_ZERO(&ReadFds);

        for (i = 0; i < readfds->fd_count; i++)
        {
            if (MainUpCallTable.lpWPUQuerySocketHandleContext(
                (Read[i].ClientSocket = readfds->fd_array[i]),
                (LPDWORD) &SocketContext, lpErrno) ==
                SOCKET_ERROR)
                return SOCKET_ERROR;
            FD_SET((Read[i].ProvSocket =
                SocketContext->ProviderSocket), &ReadFds);
        }
    }

    // Build the WriteFds set for the lower provider.
    // This is just like the ReadFds set above.
    ...

    // Build the ExceptFds set for the lower provider.
    // This is also like the ReadFds set above.
    ...

    // Call the lower provider's WSPSelect
    Ret = NextProcTable.lpWSPSelect(nfds,
        (readfds ? &ReadFds : NULL),
        (writefds ? &WriteFds : NULL),
        (exceptfds ? &ExceptFds : NULL), timeout, lpErrno);

    if (Ret != SOCKET_ERROR)
    {
        HandleCount = Ret;

        // Set up calling provider's readfds set
        if (readfds)
        {
            count = readfds->fd_count;
            FD_ZERO(readfds);

            for(i = 0; (i < count) && HandleCount; i++)
            {
                if (MainUpCallTable.lpWPUFDIsSet(
```

```
                    Read[i].ProvSocket, &ReadFds))
                {
                    FD_SET(Read[i].ClientSocket, readfds);
                    HandleCount--;
                }
            }
        }

        // Set up calling provider's writefds set.
        // This is just like the readfds set above.
        ...
        // Set up calling provider's exceptfds set.
        // This is also like the readfds set above.
        ...
    }

    return Ret;
}
```

WSAAsyncSelect

The *WSAAsyncSelect* I/O model involves managing Windows message-based notification of network events on a socket. SPI clients use this model by calling the *WSPAsyncSelect* function, which is defined as

```
int WSPAsyncSelect(
    SOCKET s,
    HWND hWnd,
    unsigned int wMsg,
    long lEvent,
    LPINT lpErrno
);
```

The *s* parameter represents the SPI client's socket that expects windows message notification of network events. The *hWnd* parameter identifies the window handle that should receive the message defined in the *wMsg* parameter when a network event (defined in the *lEvent* parameter) occurs on the socket *s.* The *lpErrno* parameter receives a Winsock error code if your implementation of this function returns *SOCKET_ERROR*. The network events your provider must support (identified in the *lEvent* parameter) are described in Table 8-3 in "The *WSAAsyncSelect* Method" section.

When an SPI client calls *WSPAsyncSelect*, your provider is responsible for notifying the SPI client when network events occur on a socket using the client's supplied window handle and the client's supplied window message that is passed through the *WSPAsyncSelect* call. Your provider can notify an SPI client of network events by calling *WPUPostMessage*, which is defined as

```
BOOL WPUPostMessage(
    HWND hWnd,
    UINT Msg,
    WPARAM wParam,
    LPARAM lParam
);
```

Your provider uses the *hWnd* parameter to identify the SPI client's window handle to be signaled. The *Msg* parameter identifies the user-defined message that was originally passed into the *wMsg* parameter of *WSPAsyncSelect*. The *wParam* parameter accepts the socket handle where a network event has occurred. The final parameter, *lParam*, takes two pieces of information. The low word of *lParam* accepts the network event that has occurred. The high word of *lParam* accepts a Winsock error code if an error has occurred in your provider implementation with respect to the network event identified in the low word of *lParam*.

If you're developing a non-IFS layered service provider, your provider is a client of a lower provider's *WSPAsyncSelect* function. As a result, your provider is required to translate an SPI client's socket handles to your provider's socket handles using *WPUQuerySocketHandleContext* before calling the lower provider's *WSPAsyncSelect* function. Since a service provider is required to inform an SPI client's window of network events using *WPUPostMessage*, your provider must intercept these window messages from the lower provider. This is a requirement because *WPUPostMessage* passes the lower provider's socket handle back in the high word of *lParam*. As a result, your provider must translate the socket handle in the high word of *lParam* to the SPI client's socket handle that was used in the originating *WSPAsyncSelect* call.

The best way to manage the interception of window messages from a lower provider is to establish a worker thread that creates a hidden window to manage network event window messages. When your provider calls *WSPAsyncSelect* on the lower provider, you simply pass the worker window handle to the lower provider's *WSPAsyncSelect* call. From there, as your provider's worker window receives network event window messages from the lower provider, your provider can inform the SPI client of the network events using *WPUPostMessage*.

WSAEventSelect

The *WSAEventSelect* I/O model involves signaling event objects when network events occur on a socket. SPI clients use this model by calling the *WSPEventSelect* function, which is defined as

```
int WSPEventSelect(
    SOCKET s,
    WSAEVENT hEventObject,
    long lNetworkEvents,
    LPINT lpErrno
);
```

The *s* parameter represents the SPI client's socket that expects notification of network events. The *hEventObject* parameter is a *WSAEVENT* object handle that your provider signals when network events specified in *lNetworkEvents* occur on the socket *s*. The *lpErrno* parameter receives a Winsock error code if your implementation of this function returns *SOCKET_ERROR*. The network events that your provider must support (identified in the *lNetworkEvents* parameter) are the same as those specified in the *WSAAsyncSelect* I/O model.

Implementing *WSPEventSelect* in a non-IFS layered provider is actually trivial. As an SPI client passes down an event object, your provider only needs to translate the SPI client's socket handle to the lower provider's socket handle using *WPUQuerySocketHandleContext*. After socket handle translation, you can call the lower provider's *WSPEventSelect* function using the event object directly from the SPI client. When I/O operations occur on the SPI client's event object, the lower provider directly signals the event object, and the SPI client is notified of the arrival of the network events. When the lower provider signals the event object, no socket handle is returned to the SPI client upon event completion. Therefore, your provider does not have to perform socket handle translation for the SPI client. This is unlike the *WSAAsyncSelect* model described earlier, in which your provider has to

perform socket handle translation on a completed network event because a socket handle is passed in the window message sent from the lower provider.

If you're developing a base provider, your provider is responsible for notifying the SPI client when network events specified in *INetworkEvents* occur on a socket using the client's supplied event object that is passed through the *WSPEventSelect* call. Your provider can notify an SPI client of network events by using the upcall *WPUSetEvent*, which is defined as

```
BOOL WPUSetEvent (
    WSAEVENT hEvent,
    LPINT lpErrno
);
```

The *hEvent* parameter represents the event object handle your provider must signal that is passed in from *WSPEventSelect*. The *lpErrno* parameter returns with a specific Winsock error code if this function returns *FALSE*.

Overlapped I/O

The overlapped I/O model requires that your provider implement an overlapped I/O manager to service both event object_based and completion routine_based overlapped I/O requests. The following SPI functions use Win32 overlapped I/O in Winsock:

- *WSPSend*
- *WSPSendTo*
- *WSPRecv*
- *WSPRecvFrom*
- *WSPIoctl*

Each function features three common parameters: an optional pointer to a *WSAOVERLAPPED* structure, an optional pointer to a *WSAOVERLAPPED_COMPLETION_ROUTINE* function, and a pointer to a *WSATHREADID* structure identifying the application thread performing the call.

The *WSAOVERLAPPED* structure is key to how a service provider communicates with an SPI client during overlapped I/O operations and is defined as

```
typedef struct _WSAOVERLAPPED {
    DWORD    Internal;
    DWORD    InternalHigh;
    DWORD    Offset;
    DWORD    OffsetHigh;
    WSAEVENT hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

Your service provider's overlapped I/O manager is responsible for managing the *Internal* field of an SPI client's *WSAOVERLAPPED* structure. At the start of overlapped processing, your service provider must set the *Internal* field to *WSS_OPERATION_IN_PROGRESS*. This is important because if the SPI client calls *WSPGetOverlappedResult* while your provider is servicing a pending overlapped operation, the *WSS_OPERATION_IN_PROGRESS* value can be used in your provider's implementation of *WSPGetOverlappedResult* to determine whether an overlapped operation is still in progress.

When an I/O operation is complete, your provider sets the *OffsetHigh* and *Offset* fields. *OffsetHigh* is set to a Winsock error code resulting from the operation, and *Offset* should be set to the flags resulting from a *WSPRecv* and *WSPRecvFrom* I/O operation. After setting these fields, your provider notifies the SPI client of the completed overlapped request through either an event object or a completion routine, depending on how the above I/O functions used the optional *WSAOVERLAPPED* structure.

Events

In event-based overlapped I/O, an SPI client invokes one of the I/O functions mentioned earlier with a *WSAOVERLAPPED* structure containing an event object. *WSAOVERLAPPED_COMPLETION_ROUTINE* must also be set to the *NULL* value. Your service provider is responsible for managing the overlapped I/O request. When it completes, your provider must alert the caller's thread of the completed I/O request by calling the upcall *WPUCompleteOverlappedRequest*, which is defined as

```
WSAEVENT WPUCompleteOverlappedRequest(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwError,
    DWORD cbTransferred,
    LPINT lpErrno
);
```

The *s* parameter and the *lpOverlapped* parameter represent the originating socket and the *WSAOVERLAPPED* structure that were originally sent down from the client. Your provider should set the *dwError* parameter to the completion status of the overlapped I/O request and the *cbTransferred* parameter to the number of bytes that were transferred during the overlapped operation. The final parameter, *lpErrno*, reports a Winsock error code if this function call returns *SOCKET_ERROR*. When *WPUCompleteOverlappedRequest* completes, it sets two fields in the SPI client's *WSAOVERLAPPED* structure: *InternalHigh* is set to the *cbTransferred* byte count, and *Internal* is set to a value other than *WSS_OPERATION_IN_PROGRESS*.

In event-based overlapped I/O, an SPI client eventually calls *WSPGetOverlappedResult* to retrieve the result of a completed overlapped request, which is defined as

```
BOOL WSPGetOverlappedResult (
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags,
    LPINT lpErrno
);
```

When *WSPGetOverlappedResult* is called, your service provider has to report back the operating state of the original overlapped request. As we mentioned earlier, your provider is responsible for managing the *Internal*, *InternalHigh*, *Offset*, and *OffsetHigh* fields in the SPI client's *WSAOVERLAPPED* structure. When *WSPGetOverlappedResult* is called, your provider should first check the *Internal* field of the SPI client's *WSAOVERLAPPED* structure. If the *Internal* field is set to *WSS_OPERATION_IN_PROGRESS*, your provider is still processing an overlapped request. If the *fWait* parameter of *WSPGetOverlappedResult* is set to *TRUE*, your provider must wait for the overlapped operation to complete by waiting on the event handle passed in the SPI client's *WSAOVERLAPPED* structure before returning the results. If the *fWait* parameter is set to *FALSE*, your provider should return the Winsock error *WSA_IO_INCOMPLETE*. Once the overlapped operation is complete or completes after waiting for results, your provider should set the parameters of *WSPGetOverlappedResult* as follows:

- *lpcbTransfer* set to the value of the *InternalHigh* field from the *WSAOVERLAPPED* structure, which reports

the number of bytes transferred by a send or receive operation

- *lpdwFlags* set to the value of the *Offset* field from the *WSAOVERLAPPED* structure, which reports any flags resulting from a *WSPRecv* or a *WSPRecvFrom* operation
- *lpErrno* set to the value of the *OffsetHigh* field from the *WSAOVERLAPPED* structure, which reports a resulting error code

Figure 14-5 demonstrates one possible way to implement *WSPGetOverlappedResult*.

Figure 14-5. WSPGetOverlappedResult *implementation details*

```
BOOL WSPAPI WSPGetOverlappedResult(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags,
    LPINT lpErrno)
{
    DWORD Ret;

    if (lpOverlapped->Internal != WSS_OPERATION_IN_PROGRESS)
    {
        *lpcbTransfer = lpOverlapped->InternalHigh;
        *lpdwFlags = lpOverlapped->Offset;
        *lpErrno = lpOverlapped->OffsetHigh;

        return(lpOverlapped->OffsetHigh == 0 ? TRUE : FALSE);
    }
    else
    {
        if (fWait)
        {
            Ret = WaitForSingleObject(lpOverlapped->hEvent,
                INFINITE);
            if ((Ret == WAIT_OBJECT_0) &&
                (lpOverlapped->Internal !=
                WSS_OPERATION_IN_PROGRESS))
            {
                *lpcbTransfer = lpOverlapped->InternalHigh;
                *lpdwFlags = lpOverlapped->Offset;
                *lpErrno = lpOverlapped->OffsetHigh;

                return(lpOverlapped->OffsetHigh == 0 ? TRUE :
                    FALSE);
            }
            else
                *lpErrno = WSASYSCALLFAILURE;
        }
        else
            *lpErrno = WSA_IO_INCOMPLETE;
    }
```

```
    return FALSE;
}
```

Completion routines

In completion routine_based overlapped I/O, an SPI client invokes one of the I/O functions mentioned earlier with a *WSAOVERLAPPED* structure and a *WSAOVERLAPPED_COMPLETION_ROUTINE* pointer. Your service provider is responsible for managing the overlapped I/O request. When the request completes, your provider must alert the caller's thread of the completed I/O by using the Win32 asynchronous procedure call (APC) I/O mechanism. The APC mechanism requires the caller's thread to be in an alertable wait state (described in Chapter 8). When your service provider is finished servicing a completion routine_based overlapped request, it must alert the SPI client of the completion by calling the upcall *WPUQueueApc*, which is defined as

```
int WPUQueueApc(
    LPWSATHREADID lpThreadId,
    LPWSAUSERAPC lpfnUserApc,
    DWORD dwContext,
    LPINT lpErrno
);
```

The *lpThreadId* parameter represents the SPI client's *WSATHREADID* structure, which is passed from an originating I/O call that supplies a completion routine. The *lpfnUserApc* parameter represents a pointer to a *WSAUSERAPC* function that serves as an intermediate function your provider must supply for callback to the SPI client. As an intermediate function, it must call the SPI client's *WSAOVERLAPPED_ COMPLETION_ROUTINE* supplied in the original overlapped call. A *WSAUSERAPC* intermediate function prototype is defined as

```
typedef void (CALLBACK FAR * LPWSAUSERAPC)(DWORD dwContext);
```

Notice that this function definition contains only one parameter: *dwContext*. When the SPI client calls this callback function, the *dwContext* parameter contains the information that was originally passed in the *dwContext* parameter of *WPUQueueApc*. Essentially, *dwContext* allows you to pass a data structure containing any information elements that you will need when you call the SPI client's *WSAOVERLAPPED_ COMPLETION_ROUTINE*, which is described in Chapter 8 as

```
void CALLBACK CompletionROUTINE(
    IN DWORD dwError,
    IN DWORD cbTransferred,
    IN LPWSAOVERLAPPED lpOverlapped,
    IN DWORD dwFlags
);
```

Your provider should pass the following information through the *dwContext* parameter of *WPUQueueApc*:

- The status of the overlapped operation as a Winsock error code
- The number of bytes transferred through the overlapped operation
- The caller's *WSAOVERLAPPED* structure

- The caller's flags passed into the I/O call

With this information, your provider can successfully call the SPI client's *WSAOVERLAPPED_COMPLETION_ROUTINE* from your intermediate completion routine.

Completion ports

The implementation of the completion port I/O model in Winsock 2 resides in the Ws2_32.dll module. As we mentioned in Chapter 8, the completion port model relies on the overlapped I/O model using event-based overlapped I/O. Therefore, no special considerations are necessary for your service provider to manage the completion port model.

Managing overlapped I/O

When an SPI client calls your layered provider using any of the overlapped I/O models mentioned above, your provider's overlapped manager should call the lower provider using either the event-based or the completion port overlapped I/O method described in Chapter 8. If your provider is running on Windows NT or Windows 2000, we recommend using completion ports to perform overlapped I/O on the lower provider. On Windows 95 and Windows 98, your only choice is to use event-based overlapped I/O. The following three upcalls are available to your provider for working with overlapped events:

```
WSAEVENT WPUCreateEvent(LPINT lpErrno);
BOOL WPUResetEvent(WSAEVENT hEvent, LPINT lpErrno);
BOOL WPUCloseEvent(WSAEVENT hEvent, LPINT lpErrno);
```

The *WPUCreateEvent* function creates and returns an event object. This function is exactly like the *WSACreateEvent* function described in Chapter 8: it creates an event object that is in manual reset mode. If *WPUCreateEvent* fails to create an event object, *NULL* is returned and *lpErrno* will contain a specific Winsock error code. The *WSPResetEvent* function is just like the *WSAResetEvent* function: it resets an event object (parameter *hEvent*) from the signaled state to the nonsignaled state. The *WPUCloseEvent* function is just like *WSACloseEvent* and frees all operating resources associated with an event object handle.

On the companion CD, our LSP example uses event-based overlapped I/O to manage all overlapped I/O activities for an SPI client. This allows the LSP sample to function on Windows 2000, Windows NT, Windows 98, and Windows 95. However, it is important to realize this example has only one thread for servicing overlapped I/O operations, limiting our provider to servicing no more than *WSA_MAXIMUM_WAIT_EVENTS* (64) event objects at a time using event-based overlapped I/O (described in Chapter 8). If your provider expects to service more than 64 event objects, you can create more servicing threads to service more event objects. If you're developing your provider for Windows NT and Windows 2000, we recommend using completion ports instead of event-based overlapped I/O to avoid this limitation.

Extension Functions

The Winsock library Mswsock.lib provides applications with extension functions that enhance Winsock functionality. Table 14-2 defines the extension functions currently supported.

Table 14-2. *Winsock extension functions*

Extension Function	*GUID*
AcceptEx	*WSAID_ACCEPTEX*
GetAcceptExSockaddrs	*WSAID_GETACCEPTEXSOCKADDRS*
TransmitFile	*WSAID_TRANSMITFILE*
WSARecvEx	Does not have an associated GUID

When an application linked to Mswsock.lib uses *AcceptEx*, *GetAcceptExSockaddrs*, and *TransmitFile*, it implicitly calls your provider's *WSPIoctl* function using the *SIO_GET_EXTENSION_FUNCTION_POINTER* option. *WSPIoctl* is

defined as

```
int WSPIoctl(
    SOCKET s,
    DWORD dwIoControlCode,
    LPVOID lpvInBuffer,
    DWORD cbInBuffer,
    LPVOID lpvOutBuffer,
    DWORD cbOutBuffer,
    LPDWORD lpcbBytesReturned,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
    LPWSATHREADID lpThreadId,
    LPINT lpErrno
);
```

When the call occurs, the *dwIoControlCode* parameter is set to the *SIO_GET_EXTENSION_FUNCTION_POINTER* value. The *lpvInBuffer* parameter contains a pointer to a globally unique identifier (GUID) that identifies which extension function the Mswsock.lib is looking for. (The GUID values listed in Table 14-2 define the currently supported extension functions.) If this GUID value matches any of the defined values, your provider must return a function pointer of your provider's implementation of the extension function through *lpvOutBuffer*. The remaining parameters do not apply directly to managing extension functions in the SPI.

We noted that the *WSARecvEx* function does not have a GUID in Table 14-2. This is because *WSARecvEx* does not invoke *WSPIoctl.* Instead, it directly calls *WSARecv*. As a result, your provider cannot directly monitor the *WSARecvEx* extension function.

Installing Transport Service Providers

Installing transport service providers involves developing a simple application to insert a layered or a base provider into the Winsock 2 catalog of service providers. How a transport provider is installed determines whether it is a layered provider or a base provider. The installation program simply configures your transport provider in the Winsock 2 system configuration database, which is a catalog of all installed service providers. The configuration database lets Winsock 2 know that your service provider exists and defines the type of service you're providing. Winsock 2 uses the database to determine which transport service providers it needs to load when a Winsock application creates a socket. Ws2_32.dll searches the database for the first provider that matches socket input parameters of the *socket* and *WSASocket* API calls, such as address family, type of socket, and protocol. Once an appropriate matching entry is found, Ws2_32.dll loads the appropriate service provider DLL that is specified in the catalog.

Essentially four SPI configuration functions are needed to successfully install and manage a service provider entry into the Winsock 2 service provider database. Each function begins with the WSC prefix:

- *WSCEnumProtocols*
- *WSCInstallProvider*
- *WSCWriteProviderOrder*
- *WSCDeInstallProvider*

These functions query and manipulate the database using a *WSAPROTOCOL_INFOW* structure, which is described in Chapter 5 in "Winsock 2 Protocol Information." For the purpose of installing a transport service provider, we are primarily concerned with the *ProviderId*, *dwCatalogEntryId*, and *ProtocolChain* fields of this structure. The *ProviderId* field is a GUID that uniquely allows you to define and install a provider on any system. The *dwCatalogEntryId* field simply identifies each *WSAPROTOCOL_INFOW* catalog entry structure in the database with a unique numeric value. The *ProtocolChain* field determines whether a *WSAPROTOCOL_INFOW* structure is a

catalog entry for a base provider, a layered provider, or a provider protocol chain. The *ProtocolChain* field is a *WSAPROTOCOLCHAIN* structure that is defined as

```
typedef struct {
    int ChainLen;
    DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN, FAR * LPWSAPROTOCOLCHAIN;
```

The *ChainLen* field determines whether a catalog entry represents a layered provider (*ChainLen* = 0), a base provider (*ChainLen* = 1), or a protocol chain (*ChainLen* > 1). A *protocol chain* is a special catalog entry that defines how to position a layered service provider between Winsock and other service providers. (See Figure 14-2.) Layered and base providers have only a single catalog entry per provider in the database. The final field, *ChainEntries*, is an array of catalog IDs used to describe the order in which to load service providers in a protocol chain. When Ws2_32.dll searches the catalog for an appropriate service provider during socket creation, it looks only for protocol chain and base provider catalog entries. Layered provider catalog entries (where *ChainLen* is 0) are ignored by Ws2_32.dll and exist only to identify a layered provider to a protocol chain in protocol chain catalog entries.

Base provider installation

To install a base provider, you have to create a *WSAPROTOCOL_INFOW* catalog entry structure that represents the base provider. This requires populating most of the fields in this structure with protocol attribute information describing the base provider. Remember to set the *ProtocolChain* structure's *ChainLen* field to the value 1. Once the structure is defined you need to install it in the catalog using the *WSCInstallProvider* function, which is defined as

```
int WSCInstallProvider(
    const LPGUID lpProviderId,
    const LPWSTR lpszProviderDllPath,
    const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
    DWORD dwNumberOfEntries,
    LPINT lpErrno
);
```

The *lpProviderId* parameter is a GUID that allows you to identify a provider to the Winsock catalog. The *lpszProviderDllPath* parameter is a string containing the load path to the provider's DLL. The string can include environment variables such as %SystemRoot%. The *lpProtocolInfoList* parameter represents an array of *WSAPROTOCOL_INFOW* data structures to install in the catalog. For the purposes of installing a base provider, you can simply assign the *WSAPROTOCOL_INFOW* structure to the first element of the array. The *dwNumberOfEntries* parameter contains the number of entries in the *lpProtocolInfoList* array. The *lpErrno* parameter contains specific error code information if this function fails with *SOCKET_ERROR*.

Layered provider installation

If you're installing a layered service provider, you need to create two *WSAPROTOCOL_INFOW* catalog entry structures. One will represent your layered provider (for example, the protocol chain length equals 0), and the other will represent a protocol chain (for example, the protocol chain length is greater than 1) linking your layered provider to a base provider. These two structures should be initialized with properties of an existing service provider's *WSAPROTOCOL_INFOW* catalog entry structure that can be retrieved by calling *WSCEnumProtocols*, which is defined as

```
int WSCEnumProtocols(
    LPINT lpiProtocols,
    LPWSAPROTOCOL_INFOW lpProtocolBuffer,
    LPDWORD lpdwBufferLength,
    LPINT lpErrno
);
```

The *lpiProtocols* parameter is an optional array of values. If *lpiProtocols* is *NULL*, information on all available protocols is returned; otherwise, information is retrieved only for those protocols listed in the array. The *lpProtocolBuffer* parameter is an application-supplied buffer that is filled in with *WSAPROTOCOL_INFOW* structures from the Winsock 2 catalog. The *lpdwBufferLength* parameter on input is the count of bytes in the *lpProtocolBuffer* buffer passed to *WSCEnumProtocols*. On output, it receives the minimum buffer size that can be passed to *WSCEnumProtocols* to retrieve all the requested information. The *lpErrno* parameter contains specific error information if this function fails with *SOCKET_ERROR*. Once you have the catalog entry of the provider that you're going to layer over, you can copy the properties of the provider's *WSAPROTOCOL_INFOW* into your newly created structures.

After initialization, you first need to install your layered provider catalog entry using *WSCInstallProvider* and then retrieve the catalog ID that gets assigned to this structure after installation by enumerating the catalog entries using *WSCEnumProtocols*. The catalog entry can then be used in setting up a protocol chain catalog entry linking your layered provider to another provider. Next *WSCInstallProvider* is called to install the chained provider. This process is illustrated in the following pseudo-code:

```
...
WSAPROTOCOL_INFOW LayeredProtocolInfoBuff,
                  ProtocolChainProtoInfo,
                  BaseProtocolInfoBuff;
...
// Retrieve BaseProtocolInfoBuff using WSCEnumProtocols()

memcpy (&LayeredProtocolInfoBuff , &BaseProtocolInfoBuff,
    sizeof(WSAPROTOCOL_INFO));
LayeredProtocolInfoBuff.dwProviderFlags = PFL_HIDDEN;
LayeredProtocolInfoBuff.ProviderId = LayeredProviderGuid;

// This entry will be filled in by the system
LayeredProtocolInfoBuff.dwCatalogEntryId = 0;

LayeredProtocolInfoBuff.ProtocolChain.ChainLen =
    LAYERED_PROTOCOL;

WSCInstallProvider(&LayeredProviderGuid, L"lsp.dll",
    &LayeredProtocolInfoBuff, 1, & install_error);

// Determine the catalog ID of the layered provider
// using the WSCEnumProtocols() function
for (i = 0; i < TotalProtocols; i++)
    if (memcmp (&ProtocolInfo[i].ProviderId, &ProviderGuid,
        sizeof (GUID))==0)
   {
      LayeredCatalogId = ProtocolInfo[i].dwCatalogEntryId;
      break;
```

```
   }

Memcpy(&ProtocolChainProtoInfo, &BaseProtocolInfoBuff,
       sizeof(WSAPROTOCOL_INFO));
ProtocolChainProtoInfo.ProtocolChain.ChainLen = 2;
ProtocolChainProtoInfo.ProtocolChain.ChainEntries[0] =
    LayeredProvideProtocolInfo.dwCatalogEntryId;
ProtocolChainProtoInfo.ProtocolChain.ChainEntries[1] =
    BaseProtocolInfoBuff.dwCatalogEntryId;

WSCInstallProvider(
    &ChainedProviderGuid,
    L"lsp.dll",               // lpszProviderDllPath
    &ProtocolChainProtoInfo,  // lpProtocolInfoList
    1,                        // dwNumberOfEntries
    &install_error            // lpErrno
);
```

Notice that the *PFL_HIDDEN* flag is specified in the *WSAPROTOCOL_INFOW* structure for the layered provider in the above pseudo-code. This flag ensures that the *WSAEnumProtocols* function (described in Chapter 5) does not include the catalog for the layered provider in its returned buffer.

Another important flag your installation program should manage is *XP1_IFS_ HANDLES*. Any non-IFS service provider that uses *WPUCreateSocketHandle* to create its socket handles should not set the *XP1_IFS_HANDLES* flag in its *WSAPROTOCOL_ INFOW* structure. Winsock applications should take the absence of the *XP1_ IFS_HANDLES* flag as a clue to avoid the use of *ReadFile* and *WriteFile* functions because of the performance penalty mentioned previously.

Ordering providers

Once a service provider is installed on a system, you must consider how Winsock 2 searches the database for service providers. Most Winsock applications specify which protocol they need through the parameters of a call to the *socket* and *WSASocket* functions. For example, if your application creates a socket using the address family *AF_INET* and the socket type *SOCK_STREAM*, Winsock 2 searches for the default TCP/IP protocol chain or base provider catalog entry in the database that provides this functionality. When you install a service provider using *WSCInstallProvider*, the catalog entry automatically becomes the last entry in the database. To make the service provider the default TCP/IP provider, you must reorder the providers in the database and place the protocol chain catalog entry ahead of other TCP/IP providers by calling *WSCWriteProviderOrder*, which is defined as

```
int WSCWriteProviderOrder(
    LPDWORD lpwdCatalogEntryId,
    DWORD dwNumberOfEntries
);
```

The *lpwdCatalogEntryId* parameter accepts an array of catalog IDs that identify how the catalog should be ordered. You can retrieve the catalog IDs in the catalog by calling *WSCEnumProtocols*, as described earlier. The *dwNumberOfEntries* parameter is a count of how many catalog entries are in your array. This function returns *ERROR_SUCCESS* (0) if it is successful; otherwise, it returns a Winsock error code.

The *WSCWriteProviderOrder* function is not part of the Ws2_32.dll library. To use this function, your application must link to Sporder.lib. Also, the associated Sporder.dll module is not part of the Windows operating system. The support DLL can be found in the Microsoft Developer Network (MSDN) library. If you plan to use it, you must distribute it with your installation application. The MSDN library also provides a convenient software utility named Sporder.exe that allows you to view and reorder catalog entries in the Winsock 2 database. Figure 14-6 offers a

quick look (using Sporder.exe) at the Winsock 2 configuration after installing a layered provider on a Windows 2000 computer.

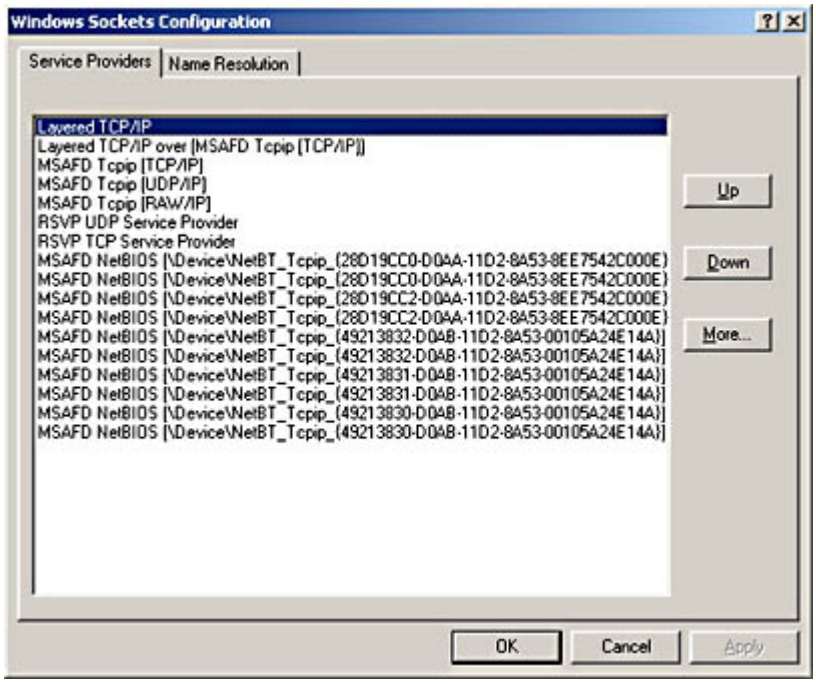

Figure 14-6. *Sporder.exe*

Removing a service provider

Removing a service provider from the Winsock 2 catalog is easy. The main task is calling the *WSCDeinstallProvider* function, which is defined as

```
int WSCDeinstallProvider(
   LPGUID lpProviderId,
   LPINT lpErrno
);
```

The *lpProviderId* parameter represents the GUID of the service provider that you're removing. The *lpErrno* parameter receives a specific Winsock error code if the function returns *SOCKET_ERROR*.

> #### Troubles with Installing Layered Providers
>
> Layered service providers have tremendous potential for value-added networking services. However, the current Winsock 2 specification does not answer an important question: how can a layered service provider know where to insert itself in a protocol chain if it finds another layered provider installed? For example, if you want to install a data encryption provider on a system that already has a URL filtering provider, obviously the data encryption provider needs to be inserted below the filtering provider in the existing protocol chain. But the problem is that the provider installation program has no way to find out what type of service an existing provider provides and therefore does not know where to insert itself into a protocol chain. This is not a big concern for a controlled networking environment in which administrators decide which providers to install and in what order to install them. But the widespread success of layered service providers is, for all intents and purposes, prohibited because the only safe installation is to install a layered service provider

directly over a base provider and make the new chain the default provider for the protocol. Such an approach guarantees the service of the new provider but removes the existing layered provider as the default provider chain.

Besides the problem of layered service provider ordering in protocol chains, there's another related issue that's not addressed in the Winsock 2 specification: how existing layered providers can protect themselves from changes in the chaining or be notified when they occur. This concern is not as grave as the first. In practice, if a layered provider protocol chain is not to be modified, the layered provider developer can hardcode the chain order within the *WSPStartup* function and install the provider as a base provider by specifying 1 in the *ProtocolChain.ChainLen* member of the LSP's *WSAPROTOCOL_INFOW* structure.

You should consider one important point when removing a service provider. There is always the possibility that a layered service provider includes your service provider's catalog ID in its protocol chain. If this is the case, you should remove your catalog ID from any protocol chains that reference your provider.

Name Space Service Providers

In Chapter 10, we showed you how an application can register and resolve services within a name space, which is an especially powerful feature for services that might be dynamically created on the network. Unfortunately, the existing name spaces available are limited in their usefulness. The Winsock 2 specification, however, provides a method for creating your own name spaces in which you can handle name registration and resolution in whatever manner you prefer.

This is accomplished by creating a DLL that implements the nine name space functions. These functions all begin with the NSP prefix and are companions to the RNR functions from Chapter 10. For example, the name space function equivalent to *WSASetService* is *NSPSetService*. After the DLL is created, it is then installed into the system catalog with a GUID that identifies the name space. Once this is done, applications can register and query services in your name space.

In this section, we'll first present how to install a name space provider, and then we'll describe the functions a name space provider must implement. Finally, we'll present a sample name space provider as well as a sample application that registers and resolves services.

Installing a Name Space Provider

A name space provider is simply a DLL that implements the name space provider functions. Before applications can use a name space, you must make the system aware of it via the *WSCInstallNameSpace* function. Conversely, once a provider is installed, you can either disable the provider or remove the provider altogether from the system catalog using the functions *WSAEnableNSProvider* and *WSAUnInstallNameSpace*, respectively. We will describe each of these functions next.

WSCInstallNameSpace

This function is used to install a provider into the system catalog and is declared as

```
int WSCInstallNameSpace (
    LPWSTR lpszIdentifier,
    LPWSTR lpszPathName,
    DWORD dwNameSpace,
    DWORD dwVersion,
    LPGUID lpProviderId
);
```

The first thing that you will notice is that all string parameters are wide character strings. Actually, all name space providers are implemented using wide character strings. We'll talk more about this later. The *lpszIdentifier* parameter is the name of the name space provider. This is the name that is enumerated when you call *WSAEnumNameSpaceProviders*, which we saw in Chapter 10. The *lpszPathName* parameter is the location of the DLL. The string can include environment variables, such as %SystemRoot%. The *dwNameSpace* parameter is a numeric identifier for the name space. For example, the header file Nspapi.h defines constants for other well-known name spaces, such as *NS_SAP* for IPX SAP. The *dwVersion* parameter sets the version number for the name space. Finally, *lpProviderId* is a GUID that identifies the name space provider.

Upon success, *WSCInstallNameSpace* returns 0; upon failure, the function returns *SOCKET_ERROR*. The most common failures are *WSAEINVAL*, which indicates that a name space with that GUID already exists; and *WSAEACCESS*, which indicates that the calling process does not have sufficient privilege. Only users of the Administrators group can install a name space.

WSCEnableNSProvider

This function is used to modify the state of a name space provider. It can be used to enable or disable the provider. The function is declared as

```
int WSCEnableNSProvider (
    LPGUID lpProviderId,
    BOOL fEnable
);
```

The *lpProviderId* parameter is the GUID identifier for the name space that you want to modify. The *fEnable* parameter is a Boolean value indicating you should either enable or disable the provider. A disabled provider is unable to process queries or registrations.

Upon success, *WSCEnableNSProvider* returns 0; upon failure, the function returns *SOCKET_ERROR*. If the provider GUID is invalid, *WSAEINVAL* is returned.

WSCUninstallNameSpace

This function removes a name space provider from the catalog. The function is defined as

```
int WSCUnInstallNameSpace ( LPGUID lpProviderId );
```

The *lpProviderId* parameter is the GUID for the name space to remove. If the GUID is invalid, the function fails with *WSAEINVAL*.

Implementing a Name Space

A name space must implement all nine name space functions that map to the RNR functions covered in Chapter 10. In addition to implementing these functions, you must also develop a method for persisting the data. That is, you must maintain the data beyond the instance of the DLL. Each process that loads the DLL receives its own data segment, which means that data stored within the DLL cannot be shared between instances. (Actually, it is possible to share information between multiple applications that have loaded the DLL, but this practice is discouraged.) See *Programming Applications for Microsoft Windows, 4th ed.*, by Jeffrey Richter (Microsoft Press, 1999) for more information on DLLs. Remember from Chapter 10 that there are three types of name spaces: dynamic, persistent, and static. Obviously, implementing a static name space might not be a good idea since it disallows programmatic registration of services. Later in this chapter, we'll present some ideas on how to maintain the data that the name space needs to persist.

You must also understand the importance of using wide character strings in all name space provider functions. This not only includes string parameters to functions but also strings within the RNR structures, such as *WSAQUERYSET* and *WSASERVICECLASSINFO*. You might be wondering how this is possible since when an application registers or resolves a name it can use either the normal (ASCII) or the wide character (UNICODE) version of the RNR functions and structure. The answer is that either version works because all ASCII calls go through an intermediate layer that converts all strings to wide character strings. This is true upon function call and return. That is, if *WSAQUERYSET* is returned to the calling application—as with *WSALookupServiceNext*—any data returned by the name space provider is originally UNICODE and is converted to ASCII before returning from the function call. That said, you can see that if your application uses RNR functions, calling the wide character versions will be faster since no conversions are required.

Of the nine functions that a name space provider must implement, only seven of the functions map to Winsock 2 RNR functions, as shown in Table 14-3. The remaining two functions are for initialization and cleanup. Once the name space is installed into the system, applications can utilize the name space by specifying either the GUID under which the name space was installed or the name space identifier that is also specified during installation. An application then makes calls to the standard Winsock 2 RNR function, as described in Chapter 10. When one of these functions is called, the equivalent name space provider function is invoked. For example, when an application calls *WSAInstallServiceClass*, which references the GUID for a custom name space, the function

NSPInstallServiceClass for that provider is invoked. In the next section, we'll cover each of the name space functions.

Table 14-3. *Mapping Winsock 2 registration and name resolution functions to name space provider functions*

Winsock Function	*Equivalent Name Space Provider Function*
WSAInstallServiceClass	*NSPInstallServiceClass*
WSARemoveServiceClass	*NSPRemoveServiceClass*
WSAGetServiceClassInfo	*NSPGetServiceClassInfo*
WSASetService	*NSPSetService*
WSALookupServiceBegin	*NSPLookupServiceBegin*
WSALookupServiceNext	*NSPLookupServiceNext*
WSALookupServiceEnd	*NSPLookupServiceEnd*

NSPStartup

The *NSPStartup* function is called whenever the name space provider DLL is loaded. Your name space implementation *must* include this function, and it must be exported from the DLL. Any per-DLL data structures required for the provider to operate can be allocated when this function is called. *NSPStartup* is prototyped as

```
int NSPStartup (
    LPGUID lpProviderId,
    LPNSP_ROUTINE lpnspRoutines
);
```

The first parameter, *lpProviderId*, is the GUID for this name space provider. The *lpnspRoutines* parameter is an *NSP_ROUTINE* structure that your implementation of this function must fill out. This structure provides function pointers to the other eight name space functions that belong to your provider. The *NSP_ROUTINE* object is defined as

```
typedef struct _NSP_ROUTINE
{
    DWORD                    cbSize;
    DWORD                    dwMajorVersion;
    DWORD                    dwMinorVersion;
    LPNSPCLEANUP             NSPCleanup;
    LPNSPLOOKUPSERVICEBEGIN  NSPLookupServiceBegin;
    LPNSPLOOKUPSERVICENEXT   NSPLookupServiceNext;
    LPNSPLOOKUPSERVICEEND    NSPLookupServiceEnd;
    LPNSPSETSERVICE          NSPSetService;
    LPNSPINSTALLSERVICECLASS NSPInstallServiceClass;
    LPNSPREMOVESERVICECLASS  NSPRemoveServiceClass;
    LPNSPGETSERVICECLASSINFO NSPGetServiceClassInfo;
} NSP_ROUTINE, FAR * LPNSP_ROUTINE;
```

The first field of the structure, *cbSize*, indicates the size of the *NSP_ROUTINE* structure. The next two fields, *dwMajorVersion* and *dwMinorVersion*, are included for versioning your provider. The versioning is arbitrary and

serves no other purpose. The provider sets the rest of the entries to their respective function pointers. For example, the provider assigns its *NSPSetService* function address (no matter what it is named) to the *NSPSetService* field. The names of your provider functions can be arbitrary, but their parameters and return types must match the provider definition.

The only action required of an *NSPStartup* implementation is filling in the *NSP_ROUTINE* structure. Once the provider completes this and any other initialization routines of its own, it returns *NO_ERROR* if everything is successful. If an error occurs, the *NSPStartup* implementation returns *SOCKET_ERROR* and sets the Winsock error code. For example, if a provider attempts and fails to allocate memory, it calls *WSASetLastError* with *WSA_NOT_ENOUGH_MEMORY* as the parameter and then returns *SOCKET_ERROR*.

This might be a good time to discuss error handling in your provider's DLL. All of the functions you must implement for a provider return *NO_ERROR* upon success and *SOCKET_ERROR* upon failure. If you determine that the call fails, set the appropriate Winsock error code before returning. If you fail to do this, any application attempting to register or query services using your name space provider will report the failure of an RNR function but *WSAGetLastError* will return 0. This will cause tremendous trouble for applications that attempt to handle errors gracefully; 0 is certainly not an expected return value upon error.

NSPCleanup

This routine is called when the provider's DLL is unloaded. Within this function, you can free any memory allocated in the *NSPStartup* routine. This routine is defined as

```
int NSPCleanup ( LPGUID lpProviderId );
```

The only parameter is your name space provider's GUID. Other than cleaning up any dynamically allocated memory, you're not required to do anything in this function.

NSPInstallServiceClass

The *NSPInstallServiceClass* function maps to *WSAInstallServiceClass* and is responsible for registering a service class. *NSPInstallServiceClass* is defined as

```
int NSPInstallServiceClass (
    LPGUID lpProviderId,
    LPWSASERVICECLASSINFOW lpServiceClassInfo
);
```

The first parameter is the provider's GUID. The *lpServiceClassInfo* parameter is the *WSASERVICECLASSINFOW* structure that is being registered. Your provider has to maintain a list of service classes and has to ensure that a service class doesn't already exist using the same GUID within the *WSASERVICECLASSINFOW* structure. In the event that the GUID is already in use, the provider must return the error *WSAEALREADY*. Otherwise, the provider should maintain this service class so that other RNR operations can refer to it.

The majority of the remaining name space provider functions refer to an installed service class.

NSPRemoveServiceClass

This function is the complement of the *NSPInstallServiceClass* function and removes the specified service class. This name space function maps to *WSARemoveServiceClass*. The function is declared as

```
int NSPRemoveServiceClass (
    LPGUID lpProviderId,
    LPGUID lpServiceClassId
);
```

As in the previous function, the first parameter is the provider's GUID. The second parameter, *lpServiceClassId*, is the service class GUID that is to be removed. The provider must remove the given service class from its storage. If the service class specified by *lpServiceClassId* is not found, the provider must generate the error *WSATYPE_NOT_FOUND*.

NSPGetServiceClassInfo

The *NSPGetServiceClassInfo* function maps to the *WSAGetServiceClassInfo* function. It retrieves the *WSANAMESPACE_INFOW* structure associated with a GUID. The function is defined as

```
int NSPGetServiceClassInfo (
LPGUID lpProviderId,
LPDWORD lpdwBufSize,
LPWSASERVICECLASSINFOW lpServiceClassInfo
);
```

Again, the first parameter is the provider's GUID. The *lpdwBufSize* parameter indicates the number of bytes contained in the third parameter, *lpServiceClassInfo*. On input, the third parameter is a *WSASERVICECLASSINFOW* structure that contains the search criteria specifying which service class to return. This structure can contain either a service class name or the GUID for the service class to return. If the provider finds a match, it must return the *WSASERVICECLASSINFOW* structure in *lpServiceClassInfo* and should update *lpdwBufSize* to indicate the number of bytes being returned.

If, given the criteria, no service classes are found, the call should fail and set *WSATYPE_NOT_FOUND* as the error. Additionally, if a service class does match but the supplied buffer is too small, the provider should update the *lpdwBufSize* parameter to indicate the correct number of bytes required and the error *WSAEFAULT* should be set.

NSPSetService

The *NSPSetService* function maps to *WSASetService* and either registers or removes services from the name space. The function is defined as

```
int NSPSetService (
    LPGUID lpProviderId,
    LPWSASERVICECLASSINFOW lpServiceClassInfo,
    LPWSAQUERYSETW lpqsRegInfo,
    WSAESETSERVICEOP essOperation,
    DWORD dwControlFlags
);
```

The first parameter is the provider's GUID. The *lpServiceClassInfo* parameter is the *WSASERVICECLASSINFOW* structure to which this service belongs. The *lpqsRegInfo* parameter is the service to either register or delete depending on the operation specified in the fourth parameter, *essOperation*. The last parameter, *dwControlFlags*,

might specify the flag *SERVICE_MULTIPLE* that can modify the specified operation.

The name space provider first verifies that the supplied service class does exist. Then, depending on which operation is specified, appropriate action is taken. For a full description of valid *essOperation* values as well as the effect of *dwControlFlags* on them, see the section on service registration in Chapter 10, which discusses *WSASetService*. Your provider's *NSPSetService* function handles these flags accordingly.

If your service provider is updating or deleting a service that cannot be found, the error *WSASERVICE_NOT_FOUND* is set. If the provider is registering a service and the *WSAQUERYSETW* structure is invalid or incomplete, the provider generates the *WSAEINVAL* error.

This function is one of the most complicated name space provider functions to implement (next to *NSPLookupServiceNext*). The provider must maintain a scheme for persisting the services that can be registered and must allow the *NSPSetService* function to update this data.

NSPLookupServiceBegin

The *NSPLookupServiceBegin* function is associated with the functions *NSPLookupServiceNext* and *NSPLookupServiceEnd* and is used to initiate a query of your name space. This function maps to *WSALookupServiceBegin* and establishes the criteria for your search. This function is prototyped as

```
int NSPLookupServiceBegin (
    LPGUID lpProviderId,
    LPWSAQUERYSETW lpqsRestrictions,
    LPWSASERVICECLASSINFOW lpServiceClassInfo,
    DWORD dwControlFlags,
    LPHANDLE lphLookup
);
```

As with previous functions in this section, the first parameter is the provider's GUID. The *lpqsRestrictions* parameter is the *WSAQUERYSETW* structure that defines the parameters for the query. The third parameter, *lpServiceClassInfo*, is the *WSASERVICECLASSINFOW* structure containing the schema information for the service class in which the query is to take place. The *dwControlFlags* parameter takes zero or more flags that affect how the query is performed. Again, for information on *WSALookupServiceBegin* and the different flags that can be used, refer to Chapter 10. Note that not all of the flags make sense for every provider. For example, if your name space does not support the notion of container objects, you don't have to worry about those flags dealing with containers. (A container is simply a way of conceptually organizing the services—what constitutes a container is open to interpretation.) Finally, *lphLookup* is an output parameter, which is a handle that defines this particular query. The handle is used in the subsequent calls to *WSALookupServiceNext* and *WSALookupServiceEnd*.

When implementing *NSPLookupServiceBegin*, keep in mind that the operation cannot be canceled, and it should complete as quickly as possible. Therefore, if you need to initiate a network query, a response should not be required in order to return successfully.

The provider itself should save the query parameters and associate a unique handle with the query for later reference. In addition to saving the handle and the query, the provider should maintain state information. We'll explore the significance of this in our discussion of the next function, *NSPLookupServiceNext*.

NSPLookupServiceNext

Once a query has been initiated with *WSALookupServiceBegin*, an application calls *WSALookupServiceNext*, which in turn calls the name space provider function *NSPLookupServiceNext*. This call is what actually searches for results that match the search criteria registered for this query. The function is defined as

```
int NSPAPI WSALookupServiceNext (
    HANDLE hLookup,
    DWORD dwControlFlags,
    LPDWORD lpdwBufferLength,
    LPWSAQUERYSET lpqsResults
);
```

The first parameter, *hLookup*, is the query handle returned from *WSALookupServiceBegin*. The *dwControlFlags* parameter can be the flag *LUP_FLUSHPREVIOUS*, which indicates that the provider should discard the last result set and move to the next one. Typically an application requests that the last result set be discarded when the application cannot supply a large enough buffer for the results. The next parameter, *lpdwBufferLength*, indicates the size of the buffer passed as the last parameter, *lpqsResults*.

When *NSPLookupServiceNext* is triggered, the provider should look up the query parameters identified by the handle *hLookup*. Once the query parameters are retrieved, a search should be initiated for all registered services within the service class specified by the query that match the supplied criteria. As we mentioned in the section on *NSPLookupServiceBegin*, the state of the query should be saved. In the event of multiple matching entries, a calling process calls *WSALookupServiceNext* multiple times, and with each call your provider needs to return a data set. When there are no more matches, the provider returns the error *WSA_E_NO_MORE*. It is also possible to cancel a query in progress if the application makes a call to *WSALookupServiceEnd* from another thread while a call to *WSALookupServiceNext* is in progress. In this event, *NSPLookupServiceNext* should fail with the error *WSA_E_CANCELLED*.

NSPLookupServiceEnd

After a query has been completed, *NSPLookupServiceEnd* is called to end the query and release any underlying resources. This function is defined as

```
int NSPLookupServiceEnd ( HANDLE hLookup );
```

The single parameter to the function is *hLookup*, which is the handle to the query that is to be closed. If the given handle cannot be found (for example, if it's an invalid handle), the call must fail with the error *WSA_INVALID_HANDLE*.

Name Space Provider Example

We have covered the steps for creating your own name space, and we have touched on some of the important name-space creation issues, such as methods for data persistence. However, developing an entire name space provider can be complicated, and the rest of this section will be devoted to our example name space provider. While the example is not the fastest or most optimized code, it illustrates the topics that require the most attention. Additionally, we kept it simple so that it's easy to follow and understand.

The example provider is located on the companion CD-ROM under the Examples\Chapter14\NSP directory in the files Mynsp.h, Mynsp.cpp, and Mynsp.def. These three files make up the name space DLL. In addition to the DLL, you'll find the name space service that is a Winsock server responsible for handling requests from the DLL. This server, which maintains the service registration data, is found in the file Mynspsvc.cpp. Two additional files, Nspsvc.cpp and Printobj.cpp, are used by both the DLL and the service and contain support routines for marshaling and demarshaling data sent on a socket between the service and the DLL. Marshaling and demarshaling data will be explained later in this section. In addition to these two files, you'll find their header files, Nspsvc.h and Printobj.h, which contain the function prototypes for the support routines. Finally, the file Rnrcs.c is a modified sample from Chapter 10 that registers and looks up services in our custom name space.

In the following sections, we will discuss how our name space is implemented. First we'll give an overview of the method we chose to persist the data. This overview will be followed by an examination of how the actual name

space DLL is structured as well as how to install the name space. Then we'll cover the implementation of the name space service. Finally, we'll look at how an application performs service registrations and queries to our custom name space.

Data persistence

For our name space, we chose a separate Winsock service to maintain the name space information. In each of the name space functions implemented in the DLL, a connection is made to this service and data is transacted in order to complete the operation. For the sake of simplicity, this service runs locally (the service listens on the loopback address 127.0.0.1). In an actual implementation, our name space service's IP address would be accessible via the Registry or some other means so that when an application invoked the name space, it could connect to the service wherever it was running. For example, in the case of DNS, the IP address of the DNS server is either set statically or obtained during a DHCP request.

Of course, writing a service to maintain the information is not the only option available. You could maintain a file on the network that keeps the necessary information; however, this is probably not the best option, as performance is then bound by disk operations. One performance limitation of our sample name space is that it establishes TCP connections to the service. A production-quality implementation is more likely to use a connectionless datagram protocol such as UDP to improve performance. Of course, this would involve additional programming requirements—such as ensuring that dropped packets are retransmitted—but the overall performance gains would be considerable.

Name space DLL

Before we look at how the name space service is implemented, let's take a look at the name space DLL. Each name space provider requires a unique GUID, and ours is defined in Mynsp.h. In addition to the unique identifier, we need a simple integer identifier for our name space. This identifier can be used in the *dwNameSpace* field of the *WSAQUERYSET* structure, as you saw in Chapter 10. The GUID and name space identifier are

```
GUID MY_NAMESPACE_GUID = {0x55a2bd9e,0xbb30,0x11d2,
                          {0x91,0x66,0x00,0xa0,0xc9,0xa7,0x86,0xe8}
                         };
#define NS_MYNSP     66
```

These values are important because applications that want to use this name space must specify these values in their Winsock calls. Of course, the developer of an application can specify these values directly or retrieve them with a *WSAEnumNameSpaceProviders* call. (See Chapter 10 for more information.) Also, be aware that if an application performs an operation specifying the *NS_ALL* name provider, the operation occurs on all installed name providers. You should bear this in mind because several Windows applications, such as Microsoft Internet Explorer, perform queries on all installed name providers. Be very careful, therefore, when testing a name provider. A poorly written name provider can cause systemwide problems. Additionally, the GUID and name space identifier values are important because they are required to install the name provider.

Now let's take a look at the *NSP* functions implemented in Mynsp.cpp. For the most part, these functions are quite similar except for the startup and cleanup functions, *NSPStartup* and *NSPCleanup*. The startup function simply initializes the *NSP_ROUTINE* structure with our custom name space functions. The cleanup routine does nothing because no cleanup is necessary.

The rest of the functions require interaction with our service to either query or register data. When communication with the service is necessary, follow these steps:

1. Connect to the service (via the *MyNspConnect* function).
2. Write a 1-byte action code. This indicates to the service which action is about to take place (service registration, service deletion, query, and so on).
3. Marshal parameters and send them to the service. The type of parameters sent depends on the operation. For example, *NSPLookupServiceNext* sends the query handle to the service so that the service can resume

the query, whereas *NSPSetService* sends an entire *WSAQUERYSET* structure.

4. Read the return code. Once the service has the necessary parameters to perform the requested operation, the return code (success or failure) of the operation is returned. The file *Mynsp.h* defines two constants—*MYNSP_SUCCESS* and *MYNSP_ERROR*—for this purpose.

5. If the requested operation was a query and the return code was success, read and demarshal the results. For example, *NSPLookupServiceNext* returns a *WSAQUERYSET* structure in the event that a matching service is found.

As you can see, implementing the DLL is not overly complicated. The *NSP* functions must take the parameters and process them, which in our case is to pass this information to the name space service. After this, it is up to the service to perform the requested operation. However, we have glossed over one difficult operation that must be performed: sending data over a socket. Normally there aren't any special requirements for sending data, but in the case of sending entire data structures there are. Most of the name space functions take either a *WSAQUERYSET* or a *WSASERVICECLASSINFO* structure as a parameter. This object must be sent or received on the socket connection to the service. This presents some difficulty because these structures aren't contiguous blocks of memory. That is, they contain pointers to strings and other structures that can be located anywhere in memory, as illustrated in Figure 14-7. What you need to do is take all these pieces of memory—wherever they are—and copy them into a single buffer one after another. This is known as *marshaling data*. On the receiving end, this process has to be reversed. That is, the data read needs to be reassembled into the original structure, and the pointers have to be "fixed" so that they point to valid memory locations on the recipient's machine.

For our name space provider, we provide functions to marshal and demarshal both the *WSANAMESPACEINFO* and *WSAQUERYSET* structures. These functions are located in Nspsvc.cpp and are used by both the name space DLL and the name space service (since both sides need the ability to marshal and demarshal these structures). All four functions are self-explanatory—we won't cover them in depth here.

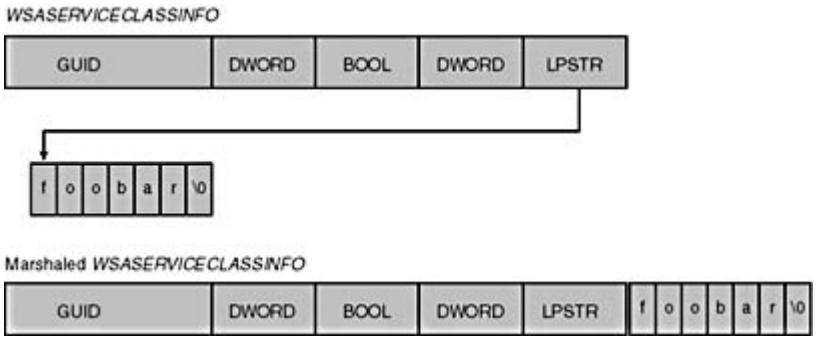

Figure 14-7. *Marshaling data*

Installing the name space

Installing a name space provider is the most trivial step in the entire process. The file *Nspinstall.c* is a simple installation program. The following code installs our provider:

```
ret = WSCInstallNameSpace(L"Custom Name Space Provider",
    L"%SystemRoot%\\System32\\Mynsp.dll", NS_MYNSP, 1,
    &MY_NAMESPACE_GUID);
if (ret == SOCKET_ERROR)
{
    printf("Failed to install name space provider: %d\n",
        WSAGetLastError());
}
```

The only parameters to the call are the name of the provider, the location of the DLL, the integer identifier, the version, and the GUID. After installation, the only requirement is to make sure the name space DLL is actually

located where you say it is. The only error that's a real possibility is trying to install a name provider with a GUID that's already in use by another provider.

Removing a name space provider is even easier. The following code snippet from our installation program removes our provider:

```
ret = WSCUnInstallNameSpace(&MY_NAMESPACE_GUID);
if (ret == SOCKET_ERROR)
{
    printf("Failed to remove provider: %d\n", WSAGetLastError());
}
```

Name space service

The name space service is the real guts of the name provider. This service keeps track of all registered service classes and service instances. When the name space DLL is triggered by a user's application, it connects to the name space service to perform the operation. The service is simple. Within the *main* function, a listening socket is established. Then, within a loop, connections are accepted from instances of the name space DLL. For the sake of simplicity, only a single connection is handled at a time. This also prevents you from having to synchronize access to the data structures that maintain the name space information. Again, a real provider would not do this because it degrades performance, but it does make the example easier to understand. Once a connection is accepted, the service reads a single byte from the name space DLL that identifies the action to follow.

Within the loop, the action is decoded and parameters are passed from the name space DLL to the service. From there, the requested actions are performed. These actions aren't complicated. The code is easy to follow, and by examining the steps for each possible action you can determine how the service works—we don't need to go into detail here. However, we will examine the structures that maintain the information. There are only two data types that name space providers are really concerned about: the *WSASERVICECLASSINFO* and *WSAQUERYSET* structures. As you have seen, the majority of the RNR functions reference one or the other of these two structures in their parameters. As a result, we maintain two global arrays—one for each structure type—along with a counter for each.

When the DLL requests to install a service class, the name service provider's *main* function first calls *LookupServiceClass*, a support routine defined in Mynspsvc.cpp. This function iterates through the array of *WSASERVICECLASSINFO* structures, *ServiceClasses*. If a service class is found with the same GUID, the service returns an error (which the DLL translates as *WSAEALREADY*). Otherwise, the new service class is added to the end of the array and the *dwNumServiceClasses* counter is incremented.

During the deletion of a service class (as when installing a service class), the *main* function calls *LookupServiceClass*. In this case however, if the service class is found, the code moves the last service class in the array to the location of the deleted class. The code then decrements the counter. One thing that is not specifically covered in the Winsock 2 specification for name space providers is what happens when a service class is to be deleted but there are still services registered that refer to it. How you choose to handle this case is up to you. Our example name space won't allow the removal of a service class if there are services registered that reference it.

The same principle that's used for maintaining *WSASERVICECLASSINFO* structures is also used for keeping track of *WSAQUERYSET* structures. There is an array of these structures named *Services* and a counter named *dwNumServices*. The addition and deletion of services is handled in the same manner as for service classes.

The last bits of information that the service must maintain are for queries. When an application initiates a query, the query parameters must be maintained for the life of the query and assigned a unique handle. This is necessary because *WSALookupServiceNext* refers to the query by the handle only. The other piece of information that must be kept is the state of the query. That is, each call to *WSALookupServiceNext* can return a single information set. The code must remember the last position within the *Services* array where data was returned so that subsequent calls to *WSALookupServiceNext* begin where the previous call left off.

Querying the name space

The last part of our name space sample is the file Rnrcs.c. This is a modified version of the name registration and resolution example presented in Chapter 10. We've made only a few changes to make the example as simple as

possible. The first change causes the code to enumerate the installed name space providers but to return only the *NS_MYNSP* provider. Second, when registering a service, Rnrcs.c enumerates only the local IP interfaces to use as the address of our service. Our service provider supports the registration of any *SOCKADDR* type. Finally, for service registration this example does not create an instance of the service; it just registers the name. Otherwise, this example behaves like the Chapter 10 example.

Running the example

Once all the examples have been compiled, installing and using the provider is simple. The following command installs the provider:

```
Nspinstall.exe install
```

Of course, don't forget to copy Mynsp.dll to %SystemRoot%\System32. Once the name space is installed, an instance of the service must be running in order to query and register services. This is done by the following command:

```
Mynspsvc.exe
```

Now you can query and register services using Rnrcs.exe. Table 14-4 shows some commands that you should execute and the order you should follow. This sequence of commands registers two services and then performs a wildcard query and a specific query. Then the command sequence queries for each of the two services and deletes them. Finally, we perform a wildcard query to illustrate that the services have been deleted.

Table 14-4. *The sequence of commands that uses the example name space to register and query services*

Command	*Meaning*
Rnrcs.exe -s:ASERVICE	Register the service ASERVICE.
Rnrcs.exe -s:BSERVICE	Register the service BSERVICE.
Rnrcs.exe -c:*	Query for all registered services.
Rnrcs.exe -c:BSERVICE	Query only for services named BSERVICE.
Rnrcs.exe -c:ASERVICE -d	Query only for services named ASERVICE, and delete them if found.
Rnrcs.exe -c:BSERVICE -d	Query only for services named BSERVICE, and delete them if found.
Rnrcs.exe -c:*	Query for all registered services.

Debug Tracing Winsock 2 SPI Functions

Winsock 2 can trace API and SPI calls invoked from Ws2_32.dll. This is extremely useful during the development phase of a transport or name space service provider. In the MSDN platform SDK, two examples, Dt_dll and Dt_dll2, are designed to trace SPI calls as they are processed on a system. The beauty of these examples is that you can modify them to trace whichever API and SPI calls you're interested in.

To use this debugging feature, you must first obtain the checked build of Winsock 2's Ws2_32.dll for your target platform. This can be found in the MSDN platform SDK under Mssdk\Bin\Debug\Winsock. Once you have the checked build, you can either copy it over the system's Ws2_32.dll under %SystemRoot%\System32 or place it in the working directory of your Winsock application. Placing the checked build in your working directory is the best choice because you won't have to modify your system for debugging purposes. After the checked build is installed, you must compile and build one of the MSDN Dt_dll samples. After compilation, you will receive a support DLL named Dt_dll.dll. You should then copy Dt_dll.dll to the working directory of your Winsock application. Once everything is in place, you can begin tracing the Winsock 2 API and SPI functions that are indirectly called from your application.

Conclusion

The Winsock 2 SPI offers software developers a method of extending the capabilities of Winsock 2 by developing a service provider. In this chapter, we explained the details of how to develop a transport provider and a name space provider. This chapter concludes our discussion of the Winsock networking technology. The next chapter deals with the Microsoft Visual Basic Winsock control. The control does use Winsock, but we won't introduce any new Winsock concepts. We simply provide a manual for how to use the Winsock control from Visual Basic. If you're not concerned with Visual Basic, you can move to the third and final section of this book, which covers Remote Access Service (RAS)—a technology that can add even greater flexibility to Winsock applications.

Chapter 15

The Microsoft Visual Basic Winsock Control

This chapter describes the Visual Basic Winsock control, a relatively new control whose purpose is to simplify the Winsock interface into an easy-to-use interface natively available from Visual Basic. Before the control was available, the only option for Winsock network programming from Visual Basic was to import all the Winsock functions from the DLL and redefine the necessary structures, which are many. This process was extremely time-consuming and prone to numerous errors, such as mismatching the type declarations. However, if you need the extra flexibility offered by directly importing Winsock into Visual Basic, take a look at the Visual Basic examples that are available throughout Part II of this book. Each Visual Basic example contains a file, Winsock.bas, that imports the necessary constants and functions. This chapter focuses only on the Visual Basic Winsock control. We'll first cover the properties and methods of the control and then present several examples that use the control.

The first Winsock control was introduced with Visual Basic 5.0. A revised version of the control became available with the Visual Studio Service Pack 2. A Service Pack 3 released later did not change the control from the SP2 version. Visual Basic 6.0 includes the latest version of the Winsock control. The various version differences are discussed toward the end of this chapter.

The Winsock control provides only a basic interface to the Winsock APIs. Unlike Winsock, which is a protocol-independent interface, the Winsock control can use only the IP transport. Additionally, the control is based on the Winsock 1.1 specification. The control supports both TCP and UDP, but in a rather limited sense. The control itself is not able to access any socket options, which means that features such as multicasting and broadcasting aren't available. Basically, the Winsock control is useful only if you require basic data networking capabilities. It does not provide the best performance because it buffers data within the control before it passes it to the system, thus adding a bit of overhead and uncertainty.

Properties

Now that you have an idea of what functionality the control provides, let's look at the properties exposed by the control. Table 15-1 contains a list of the properties available for affecting the control's behavior and for obtaining information about the control's state.

Table 15-1. *Winsock control properties*

Property Name	*Return Value*	*Read-Only?*	*Description*
BytesReceived	*Long*	Yes	Returns the number of bytes pending in the receive buffer. Use the *GetData* method to retrieve the data.
LocalHostName	*String*	Yes	Returns the local machine name.
LocalIP	*String*	Yes	Returns a string of the dotted decimal IP address of the local machine.
LocalPort	*Long*	No	Returns or sets the local port to use. Specifying 0 for the port tells the system to randomly choose an available port. Generally, only a client uses 0.
Protocol	*Long*	No	Returns or sets the protocol for the control, which supports either TCP or UDP. The constant values to set are *sckTCPProtocol* and *sckUDPProtocol*, which correspond to 0 and 1 respectively.
RemoteHost	*String*	No	Returns or sets the remote machine name. You can use either the string host name or the dotted decimal string representation.
RemoteHostIP	*String*	Yes	Returns the IP address of the remote machine. For TCP connections, this field is set upon a successful connection. For UDP operations, this field is set upon the *DataArrival* event, which then contains the IP address of the sending machine.
RemotePort	*Long*	No	Returns or sets the remote port to connect to.
SocketHandle	*Long*	Yes	Returns a value that corresponds to the socket handle.
State	*Integer*	Yes	Returns the state of the control, which is an enumerated type. See Table 15-2 for the socket state constants.

After reading Chapter 7, you should be familiar with these basic properties. They are clearly analogous to the basic Winsock functions presented in the client/server examples discussed in Chapter 7. A few properties that don't relate well to the Winsock API do need to be set to use the control properly. First the *Protocol* property needs to be set in order to tell the control what type of socket you're looking for—either *SOCK_STREAM* or *SOCK_DGRAM*. The control performs the actual socket creation under the hood, and this property is the only control you have over it. The *SocketHandle* property can be read after a connection succeeds or after a server binds to wait for connections. This is useful if you want to pass the handle to other Winsock API functions imported from a DLL. The *State* property can be used to obtain information about what the control is currently doing. This is important because the control is asynchronous, and events can be fired at any time. Use this property to make sure that the socket is in a valid state for any subsequent operations. Table 15-2 contains the possible socket states and their meanings.

Table 15-2. *Socket states*

Constant	*Value*	*Meaning*
sckClosed	0	Default. Closed.
sckOpen	1	Open.
sckListening	2	Listening for connections.
sckConnectionPending	3	Connection request has arrived but has not completed yet.
sckResolvingHost	4	Host name is being resolved.
sckHostResolved	5	Host name resolution has completed.
sckConnecting	6	Connection request started but has not completed.
sckConnected	7	Connection completed.
sckClosing	8	Peer has initiated a close.
sckError	9	An error has occurred.

Methods

The Winsock control has only a handful of methods. With a couple of exceptions, most of these method names mirror their Winsock equivalents. The method to read pending data is named *GetData*. Normally you would call the *GetData* method once the *DataArrival* event is triggered, notifying you that data has arrived. The method for sending data is named *SendData*. Additionally, a method named *PeekData* is similar to calling the Winsock *recv* function with the *MSG_PEEK* option. As always, message peeking is evil and should be avoided at all costs. Table 15-3 lists the available methods with their parameters. The methods themselves will be discussed in more detail in the client and server example sections later in this chapter.

Table 15-3. *Winsock control methods*

Method	*Parameters*	*Return Value*	*Description*
Accept	*RequestID*	Void	For TCP connections only. Use this method to accept incoming connections when handling a *ConnectionRequest* event.
Bind	*LocalPort* *LocalIP*	Void	Binds the socket to the given local port and IP. Use *Bind* if you have multiple network adapters. *Bind* must be called before *Listen*.
Close	None	Void	Closes the connection or the listening socket.
Connect	*RemoteHost* *RemotePort*	Void	Establishes a TCP connection to the given *RemotePort* on the given *RemoteHost* number.
GetData	*Data* *Type* *MaxLen*	Void	Retrieves the current data pending. The *Type* and *MaxLen* parameters are optional. The *Type* parameter defines the type of the data to be read. The *MaxLen* parameter specifies how many bytes or characters to retrieve. *GetData* ignores the *MaxLen* parameter for types other than byte array and string.
Listen	None	Void	Creates a socket and places it in listen mode. *Listen* is used only for TCP connections.
PeekData	*Data* *Type* *MaxLen*	Void	Behaves exactly like *GetData* except that the data is not removed from the system's buffer.
SendData	*Data*	Void	Sends data to the remote computer. If a UNICODE string is passed, it will be converted to an ANSI string first. Always use a byte array for binary data.

Events

Events are asynchronous routines that get called upon a specific event. In your Visual Basic application, you must handle the various events that might be generated by the Winsock control in order to successfully use the control. Generally, these events are triggered by actions that the peer initiates. For example, the TCP half-close is triggered when one side of a TCP connection closes the socket. The side initiating the close generates a FIN, and the peer responds with an ACK to acknowledge the close request. The peer receiving the FIN has the *Close* event triggered. This tells your Winsock application that the other side is no longer sending data. Your application then reads any remaining data and calls the *Close* method on your end to completely shut down the connection. Table 15-4 lists all possible Winsock events that can be triggered, along with a description of each event.

Table 15-4. *Winsock control events*

Event	*Arguments*	*Description*
Close	None	Occurs when the remote computer closes the connection
Connect	None	Occurs after the *Connect* method completes successfully
ConnectionRequest	*RequestID*	Occurs when a remote machine requests a connection
DataArrival	*bytesTotal*	Occurs when new data arrives
Error	*Number* *Description* *Scode* *Source* *HelpFile* *HelpContext* *CancelDisplay*	Occurs whenever a Winsock error is generated
SendComplete	None	Occurs upon completion of a send operation
SendProgress	*bytesSent* *bytesRemaining*	Occurs while data is being sent

UDP Example

Let's examine a sample UDP application. Take a look at the sample Visual Basic project SockUDP.vbp in the Chapter 15 directory on the CD. When the project is compiled and run, you will see a dialog similar to the one illustrated in Figure 15-1. This sample application is both a sender and a receiver of UDP messages, and therefore you can use just one instance to send and receive messages. Additionally, all the code behind the form, buttons, and Winsock controls is given in Figure 15-2.

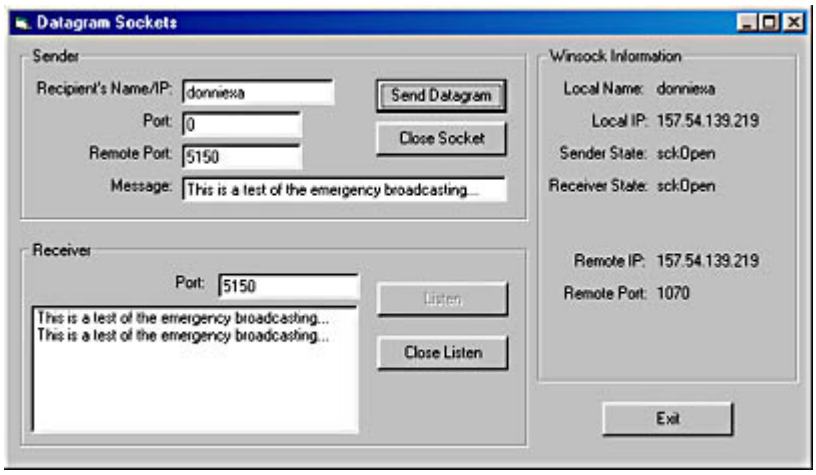

Figure 15-1. *Sample UDP application*

When you look at the form, you see two Winsock controls. One control is used to send datagrams, and the other is used to receive datagrams. You can also see three group boxes: one for the sender, one for the receiver, and one for general Winsock information. For the sender, you need somewhere to put the recipient's host name or IP address. When you set the *RemoteHost* property, you can use either the machine's textual name or a string representation of the dotted-decimal numeric IP address. The control resolves the name if needed. You also need the remote port to which you will send the UDP packets. Also, notice the text box for the local port, *txtSendLocalPort*. For the sender, it doesn't really matter which local port you send the data on, only which port you're sending to. If you leave the local port set to 0, the system assigns an unused port. The last text box, *txtSendData*, is for the string data to be sent. Additionally, there are two command buttons: one for sending the data and one for closing the socket. To send datagrams, you must bind the Winsock control to a remote address, a remote port, and a local port before you can send any data. This means that if you want to change any one of these three parameters, you need to close the socket first and then rebind to the new parameters. That is why the form has a Close Socket button.

Figure 15-2. *UDP sample code*

```
Option Explicit

Private Sub cmdExit_Click()
    Unload Me
End Sub

Private Sub cmdSendDgram_Click()
    ' If the socket state is closed, we need to bind to a
    ' local port and also to the remote host's IP address and port
    If (sockSend.State = sckClosed) Then
        sockSend.RemoteHost = txtRecipientIP.Text
        sockSend.RemotePort = CInt(txtSendRemotePort.Text)
        sockSend.Bind CInt(txtSendLocalPort.Text)
```

```
        cmdCloseSend.Enabled = True
    End If
    '
    ' Now we can send the data
    '
    sockSend.SendData txtSendData.Text
End Sub

Private Sub cmdListen_Click()
    ' Bind to the local port
    '
    sockRecv.Bind CInt(txtRecvLocalPort.Text)
    '
    ' Disable this button since it would be an error to bind
    ' twice (a close needs to be done before rebinding occurs)
    '
    cmdListen.Enabled = False
    cmdCloseListen.Enabled = True
End Sub

Private Sub cmdCloseSend_Click()
    ' Close the sending socket, and disable the Close button
    '
    sockSend.Close
    cmdCloseSend.Enabled = False
End Sub

Private Sub cmdCloseListen_Click()
    ' Close the listening socket
    '
    sockRecv.Close
    ' Enable the right buttons
    '
    cmdListen.Enabled = True
    cmdCloseListen.Enabled = False
    lstRecvData.Clear
End Sub

Private Sub Form_Load()
    ' Initialize the socket protocols, and set up some default
    ' labels and values
    '
    sockSend.Protocol = sckUDPProtocol
    sockRecv.Protocol = sckUDPProtocol

    lblHostName.Caption = sockSend.LocalHostName
    lblLocalIP.Caption = sockSend.LocalIP

    cmdCloseListen.Enabled = False
    cmdCloseSend.Enabled = False

    Timer1.Interval = 500
    Timer1.Enabled = True
```

```
End Sub

Private Sub sockSend_Error(ByVal Number As Integer, _
        Description As String, ByVal Scode As Long, _
        ByVal Source As String, ByVal HelpFile As String, _
        ByVal HelpContext As Long, CancelDisplay As Boolean)
    MsgBox Description
End Sub

Private Sub sockRecv_DataArrival(ByVal bytesTotal As Long)
    Dim data As String

    ' Allocate a string of sufficient size, and get the data;
    ' then add it to the list box
    data = String(bytesTotal + 2, Chr$(0))
    sockRecv.GetData data, , bytesTotal
    lstRecvData.AddItem data
    ' Update the remote IP and port labels
    '
    lblRemoteIP.Caption = sockRecv.RemoteHostIP
    lblRemotePort.Caption = sockRecv.RemotePort
End Sub

Private Sub sockRecv_Error(ByVal Number As Integer, _
        Description As String, ByVal Scode As Long, _
        ByVal Source As String, ByVal HelpFile As String, _
        ByVal HelpContext As Long, CancelDisplay As Boolean)
    MsgBox Description
End Sub

Private Sub Timer1_Timer()
    ' When the timer goes off, update the socket status labels
    '
    Select Case sockSend.State
        Case sckClosed
            lblSenderState.Caption = "sckClosed"
        Case sckOpen
            lblSenderState.Caption = "sckOpen"
        Case sckListening
            lblSenderState.Caption = "sckListening"
        Case sckConnectionPending
            lblSenderState.Caption = "sckConnectionPending"
        Case sckResolvingHost
            lblSenderState.Caption = "sckResolvingHost"
        Case sckHostResolved
            lblSenderState.Caption = "sckHostResolved"
        Case sckConnecting
            lblSenderState.Caption = "sckConnecting"
        Case sckClosing
            lblSenderState.Caption = "sckClosing"
        Case sckError
            lblSenderState.Caption = "sckError"
        Case Else
```

```
            lblSenderState.Caption = "unknown"
    End Select
    Select Case sockRecv.State
        Case sckClosed
            lblReceiverState.Caption = "sckClosed"
        Case sckOpen
            lblReceiverState.Caption = "sckOpen"
        Case sckListening
            lblReceiverState.Caption = "sckListening"
        Case sckConnectionPending
            lblReceiverState.Caption = "sckConnectionPending"
        Case sckResolvingHost
            lblReceiverState.Caption = "sckResolvingHost"
        Case sckHostResolved
            lblReceiverState.Caption = "sckHostResolved"
        Case sckConnecting
            lblReceiverState.Caption = "sckConnecting"
        Case sckClosing
            lblReceiverState.Caption = "sckClosing"
        Case sckError
            lblReceiverState.Caption = "sckError"
        Case Else
            lblReceiverState.Caption = "unknown"
    End Select
End Sub
```

Sending UDP Messages

Now that you know the sender's general capabilities, let's look at the code behind the scenes. First take a look at the *Form_Load* routine. The first step is to set the *Protocol* property of the *sockSend* Winsock control to UDP by using the *sckUDPProtocol* enumerated type. The other commands in this routine don't apply to the sending functionality except for disabling the *cmdCloseSend* command button. We do this for completeness because calling the *Close* method on an already closed control does nothing. Note that the default state of the Winsock control is closed.

Next look at the *cmdSendDgram_Click* routine, which is triggered by clicking on the Send Data button. This is the heart of sending a UDP message. The first step in the code is to check the socket's state. If the socket is in the closed state, the code binds the socket to a remote address, a remote port, and a local port. Once the code binds a UDP Winsock control with these parameters, the state of the control changes from *sckClosed* to *sckOpen*. If the code doesn't perform this check and attempts to bind the socket on every send, the run-time error 40020, "Invalid operation at current state," will be generated. Once a socket is bound, it remains bound until it is closed. This is why the code enables the Close Socket button for the sending socket once the control is bound. The last step is to call the *SendData* method with the data the user wants to send. When the *SendData* method returns, the code has finished sending data.

Only two other subroutines are associated with sending UDP messages. The first is *cmdCloseSend*, which as its name implies closes the sending socket, allowing the user to change the remote host, remote port, or local port parameter before sending data again. The other routine is *sockSend_Error*, which is a Winsock event. This event is triggered whenever a Winsock error is generated. Because UDP is unreliable, few errors will be generated. If an error does occur, the code simply prints out the error's description. The only message a user might see in this application is a destination unreachable message.

Receiving UDP Messages

As you can see, sending a UDP packet with the Winsock control is simple and straightforward. Receiving UDP packets is even easier. Let's go back to the *Form_Load* routine to see what needs to be done to receive a UDP

message. As we saw with the sending Winsock control, the code sets the *Protocol* property to UDP. The code also disables the Close Listen button. Again, closing an already closed socket won't hurt, but the code does it for the sake of completeness. Also, it's always a good idea to think, "What could happen if I call method X?" at different points in the program. This is the source of most of the problems developers encounter with the control: calling a method when the state of control is invalid. An example of this is calling the *Connect* method on a Winsock control that is already connected.

To listen for incoming UDP packets, let's look at the *cmdListen_Click* routine. This is the handler for the Listen button. The only necessary step is to call the *Bind* method on the receiving Winsock control, passing the local port on which the user wants to listen for incoming UDP datagrams. When listening for incoming UDP packets, the code needs only the local port—the remote port on which the data was sent is not relevant. After the code binds the control, it disables the *cmdListen* button—this prevents the possibility of the user clicking the Listen button twice. Trying to bind an already bound control will fail with a run-time error.

At this point, the *sockRecv* control is registered to receive UDP data. When the control receives UDP data on the port it's bound to, the *DataArrival* event is triggered. This event is implemented in the *sockRecv_DataArrival* routine. The parameter passed into the event, *bytesTotal*, is the number of bytes available to be read. The code allocates a string slightly larger than the amount of data being read. Then it calls the *GetData* method, passing the allocated string as the first parameter. The second parameter defaults to the Visual Basic type *vbString*, and the third parameter specifies the number of bytes that need to be read, which, in this example, is the value *bytesTotal*. If the code requests to read a smaller number of bytes than that specified by the *bytesTotal* parameter, a run-time error is generated. Once the data is read into the character buffer, the code adds it to the list box of messages read. The last few steps in this subroutine set the label captions for the remote host's IP address and port number. Upon receipt of each UDP packet, the *RemoteHostIP* and *RemotePort* properties are set to the remote host's IP address and port number for the packet just received. Therefore, if the program receives multiple UDP packets from several hosts, the values of these properties will change often.

The last two subroutines associated with receiving UDP messages are *cmdCloseListen_Click* and *sockRecv_Error*. The user invokes the *cmdCloseListen_Click* handler by clicking the Close Listen button. The routine simply calls the *Close* method on the Winsock control. Closing a UDP control frees the underlying socket descriptor. The *sockRecv_Error* event is called whenever a Winsock error is generated. As we mentioned previously in the UDP send section, few UDP errors are generated to begin with because of their unreliable nature.

Obtaining Winsock Information

The last part of our UDP example is the Winsock Information group box. The local name and local IP labels are set at form load time. As soon the form loads and Winsock controls are instantiated, the properties *LocalHostName* and *LocalIP* are set to the host name and IP address of the host machine and can be read at any time. The next two labels, Sender State and Receiver State, display the current state of the two Winsock controls used by the application. The state information is updated every half second. This is where the Timer control comes in. Every 500 milliseconds, the timer control triggers the Timer handler, which queries the socket states and updates the labels. We print the socket states for informative purposes only. The last two labels, Remote IP and Remote Port, are set whenever a UDP message is received, as discussed in the previous paragraph.

Running the UDP Example

Now that you understand how to send and receive UDP messages, let's take a look at the example as it runs. The best way to test it is to run an instance of the application on three separate machines. On one of the applications, click the Listen button. On the other two, set the Recipient's Name/IP field to the name of the machine on which the first application is running. This can be either a host name or an IP address. Now click the Send Datagram button a few times, and the messages should appear in the receiver's message window. Upon receipt of each message, the Winsock Information fields should be updated with the IP address of the sender and the port number on which the message was sent. You can even use the Sender commands on the same application as the receiver to send messages on the same machine.

Another interesting test is using either subnet-directed broadcasts or broadcast datagrams. Assuming that you're testing all three machines on the same subnet, you can send a datagram to a specified subnet and all listening applications receive the message. For example, on our test machines we have two single-homed machines with IP addresses 157.54.185.186 and 157.54.185.224. The last machine is multihomed, with IP addresses 169.254.26.113 and 157.54.185.206. As you can see, all three machines share the subnet 157.54.185.255. Let's digress for a moment to discuss an important detail. If you want to receive UDP messages, you must implicitly bind to the first IP address stored in the network bindings when you call the *Bind* method. This is sufficient if your machine has only one network card. In some cases, however, a machine has more than one network interface and therefore more than one IP address. In these cases, the second parameter to the *Bind* method is the IP address on

which to bind. Unfortunately, the Winsock control property *LocalIP* returns only one IP address, and the control provides no other method for obtaining other IP addresses associated with the local machine.

Now let's try some broadcasting. Close each sending or listening socket on each of the instances running. On the two single-homed machines, click the Listen button so that each machine can receive datagram messages. We don't use the multihomed machine because we aren't binding to any particular IP address in the code. On the third machine, enter the recipient's address as 157.54.185.255 and click on the Send Data button a few times. You should see the message being received by both listening applications. If your sending machine is also multihomed, you might be wondering how it knows which network interface to send the datagram over. It is one of the routing table's functions to determine the best interface to send the message over, given the message's destination address and the address of each interface on the local machine. If you would like to learn more about subnets and routing, consult a book on TCP/IP such as *TCP/IP Illustrated Volume 1,* by W. Richard Stevens (Addison-Wesley, 1994) or *TCP/IP: Architecture, Protocols, and Implementation with IP v6 and IP Security,* by Dr. Sidnie Feit (McGrawHill, 1996). The last test to try is to close the sender's socket on the third machine, enter the recipient's address as 255.255.255.255, and click the Send Datagram button a few times. The results should be the same: the other two listening programs should receive the message. However, the only difference on a multihomed machine is that the UDP message is being broadcast on each network attached to the machine.

UDP States

You might be a bit confused by the order in which method calls should be made to successfully send or receive datagrams. As mentioned earlier, the most common mistake when programming the Winsock control is to call a method whose operation is not valid for the current state of the control. To help alleviate this kind of mistake, take a look at Figure 15-3, which is a state diagram of the socket states when you are using UDP messages. Notice that the default starting state is always *sckClosed*, and no errors are generated for invalid host names.

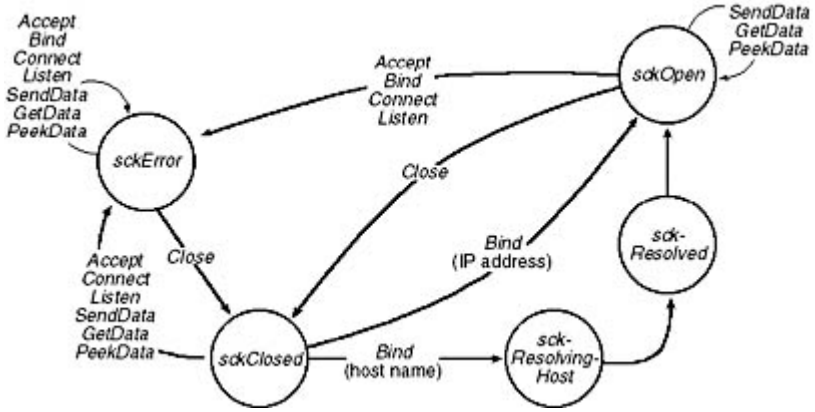

Figure 15-3. *UDP state diagram*

TCP Example

Using a Winsock control with the TCP protocol is a bit more involved and complex than using the control with its UDP counterpart. As we did with UDP, we will present a sample TCP application and go over its specifics in order to gain an understanding of the steps necessary to successfully use a TCP connection. Figure 15-4 shows the application running, and Figure 15-5 contains the sample code for this section. You can find the code for this Visual Basic project—named SockTCP.vbp—under the Chapter 15 directory.

Let's take a look at the form in Figure 15-4 to gain an understanding of this application's capabilities. Again, you'll notice three group boxes: TCP Server, TCP Client, and Winsock Information. First we'll discuss the TCP Server portion of the application. The server has a text box, *txtServerPort*, for the local port that the server will be bound to in order to listen for incoming client connections. Also, the server has two buttons, one to put the server in listening mode and the other to shut down the server and stop accepting incoming connections. Finally, the server has a single Winsock control named *sockServer*. If you take a look at the properties page, you'll see that the *Index* property has been set to 0. This means the control is actually an array capable of holding many instances of the Winsock control. The 0 signifies that at form load time only one instance (element 0 of the array) will be created. At any time we can dynamically load another instance of a Winsock control into an element of the array.

The Winsock control array is the basis of our server capabilities. Remember that a single Winsock control has only one socket handle associated with it. In Chapter 7, you learned that when a server accepts an incoming connection, a new socket is created to handle that connection. Our application is designed to dynamically load additional Winsock controls upon a client connection so that the connection can be passed to the newly loaded control without interrupting the server socket to handle the connection. Another way to accomplish this is to actually put *x* number of Winsock controls on the form at design time. However, this is wasteful and does not scale well. When the application begins, a great deal of time will be spent loading all the resources necessary for every control; there is also the issue of how many controls to use. By placing *x* number of controls, you limit yourself to *x* number of concurrent clients. If your application requirements allow for only a fixed number of concurrent connections, placing a fixed number of Winsock controls on the form will work and is probably a bit simpler than using an array. For most applications, however, an array of Winsock controls is the best way to go.

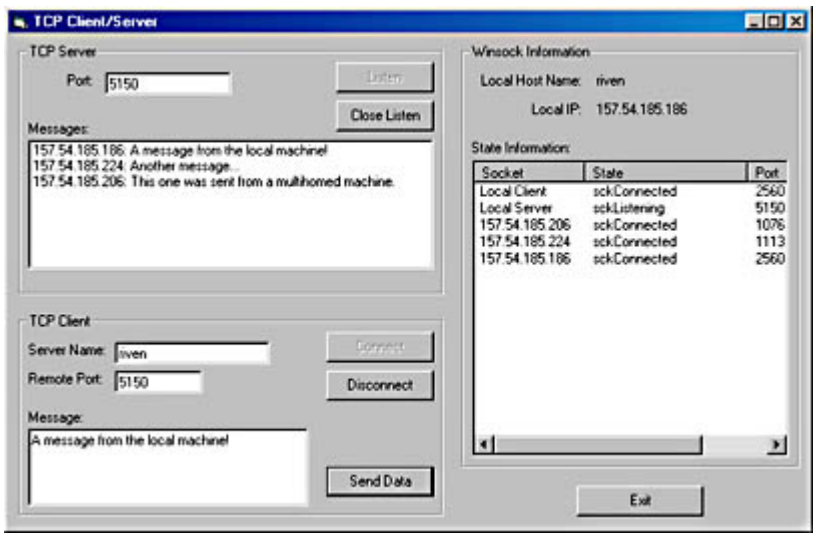

Figure 15-4. *Sample TCP application*

Figure 15-5. *Sample TCP application*

```
Option Explicit

' The index value of the last Winsock control dynamically loaded
' in the sockServer array
Private ServerIndex As Long
```

```
Private Sub cmdCloseListen_Click()
    Dim itemx As Object
    ' Close the server's listening socket. No more
    ' clients will be allowed to connect.
    '
    sockServer(0).Close
    cmdListen.Enabled = True
    cmdCloseListen.Enabled = False

    Set itemx = lstStates.ListItems.Item(2)
    itemx.SubItems(2) = "-1"
End Sub

Private Sub cmdConnect_Click()
    ' Have the client control attempt to connect to the
    ' specified server on the given port number
    '
    sockClient.LocalPort = 0
    sockClient.RemoteHost = txtServerName.Text
    sockClient.RemotePort = CInt(txtPort.Text)
    sockClient.Connect

    cmdConnect.Enabled = False
End Sub

Private Sub cmdDisconnect_Click()
    Dim itemx As Object
    ' Close the client's connection and set up the command
    ' buttons for subsequent connections
    '
    sockClient.Close

    cmdConnect.Enabled = True
    cmdSendData.Enabled = False
    cmdDisconnect.Enabled = False
    ' Set the port number to -1 to indicate no connection
    '
    Set itemx = lstStates.ListItems.Item(1)
    itemx.SubItems(2) = "-1"
End Sub

Private Sub cmdExit_Click()
    Unload Me
End Sub

Private Sub cmdListen_Click()
    Dim itemx As Object
    ' Put the server control into listening mode on the given
    ' port number
    '
    sockServer(0).LocalPort = CInt(txtServerPort.Text)
    sockServer(0).Listen
```

```
    Set itemx = lstStates.ListItems.Item(2)
    itemx.SubItems(2) = sockServer(0).LocalPort

    cmdCloseListen.Enabled = True
    cmdListen.Enabled = False
End Sub

Private Sub cmdSendData_Click()
    ' If we're connected, send the given data to the server
    '
    If (sockClient.State = sckConnected) Then
        sockClient.SendData txtSendData.Text
    Else
        MsgBox "Unexpected error! Connection closed"
        Call cmdDisconnect_Click
    End If
End Sub

Private Sub Form_Load()
    Dim itemx As Object

    lblLocalHostname.Caption = sockServer(0).LocalHostName
    lblLocalHostIP.Caption = sockServer(0).LocalIP

    ' Initialize the Protocol property to TCP since that's
    ' all we'll be using
    '
    ServerIndex = 0
    sockServer(0).Protocol = sckTCPProtocol
    sockClient.Protocol = sckTCPProtocol
    ' Set up the buttons
    '
    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
    ' Initialize the ListView control that contains the
    ' current state of all Winsock controls created (not
    ' necessarily connected or being used)
    '
    Set itemx = lstStates.ListItems.Add(1, , "Local Client")
    itemx.SubItems(1) = "sckClosed"
    itemx.SubItems(2) = "-1"
    Set itemx = lstStates.ListItems.Add(2, , "Local Server")
    itemx.SubItems(1) = "sckClosed"
    itemx.SubItems(2) = "-1"
    ' Initialize the timer, which controls the rate of refresh
    ' on the above socket states
    '
    Timer1.Interval = 500
    Timer1.Enabled = True
End Sub
```

```
Private Sub sockClient_Close()
    sockClient.Close
End Sub

Private Sub sockClient_Connect()
    Dim itemx As Object

    ' The connection was successful: enable the transfer data
    ' buttons
    cmdSendData.Enabled = True
    cmdDisconnect.Enabled = True

    Set itemx = lstStates.ListItems.Item(1)
    itemx.SubItems(2) = sockClient.LocalPort
End Sub

Private Sub sockClient_Error(ByVal Number As Integer, _
        Description As String, ByVal Scode As Long, _
        ByVal Source As String, ByVal HelpFile As String, _
        ByVal HelpContext As Long, CancelDisplay As Boolean)
    ' An error occured on the Client control: print a message,
    ' and close the control. An error puts the control in the
    ' sckError state, which is cleared only when the Close
    ' method is called.
    MsgBox Description
    sockClient.Close
    cmdConnect.Enabled = True
End Sub

Private Sub sockServer_Close(index As Integer)
    Dim itemx As Object
    ' Close the given Winsock control
    '
    sockServer(index).Close

    Set itemx = lstStates.ListItems.Item(index + 2)
    lstStates.ListItems.Item(index + 2).Text = "---.---.---.---"
    itemx.SubItems(2) = "-1"

End Sub

Private Sub sockServer_ConnectionRequest(index As Integer, _
        ByVal requestID As Long)
    Dim i As Long, place As Long, freeSock As Long, itemx As Object

    ' Search through the array to see wether there is a closed
    ' control that we can reuse
    freeSock = 0
    For i = 1 To ServerIndex
        If sockServer(i).State = sckClosed Then
            freeSock = i
            Exit For
        End If
```

```
    Next i
    ' If freeSock is still 0, there are no free controls
    ' so load a new one
    '
    If freeSock = 0 Then
        ServerIndex = ServerIndex + 1
        Load sockServer(ServerIndex)

        sockServer(ServerIndex).Accept requestID
        place = ServerIndex
    Else
        sockServer(freeSock).Accept requestID
        place = freeSock
    End If
    '  If no free controls were found, we added one above.
    '  Create an entry in the ListView control for the new
    '  control.  In either case set the state of the new
    '  connection to sckConnected.
    '
    If freeSock = 0 Then
        Set itemx = lstStates.ListItems.Add(, , _
            sockServer(ServerIndex).RemoteHostIP)
    Else
        Set itemx = lstStates.ListItems.Item(freeSock + 2)
        lstStates.ListItems.Item(freeSock + 2).Text = _
            sockServer(freeSock).RemoteHostIP
    End If
    itemx.SubItems(2) = sockServer(place).RemotePort

End Sub

Private Sub sockServer_DataArrival(index As Integer, _
        ByVal bytesTotal As Long)
    Dim data As String, entry As String

    ' Allocate a large enough string buffer and get the
    ' data
    '
    data = String(bytesTotal + 2, Chr$(0))
    sockServer(index).GetData data, vbString, bytesTotal
    ' Add the client's IP address to the beginning of the
    ' message and add the message to the list box
    '
    entry = sockServer(index).RemoteHostIP & ": " & data
    lstMessages.AddItem entry
End Sub

Private Sub sockServer_Error(index As Integer, _
        ByVal Number As Integer, Description As String, _
        ByVal Scode As Long, ByVal Source As String, _
        ByVal HelpFile As String, ByVal HelpContext As Long, _
        CancelDisplay As Boolean)
    ' Print the error message and close the specified control.
```

```
    ' An error puts the control in the sckError state, which
    ' is cleared only when the Close method is called.
    MsgBox Description
    sockServer(index).Close
End Sub

Private Sub Timer1_Timer()
    Dim i As Long, index As Long, itemx As Object

    ' Set the state of the local client Winsock control
    '
    Set itemx = lstStates.ListItems.Item(1)
    Select Case sockClient.State
        Case sckClosed
            itemx.SubItems(1) = "sckClosed"
        Case sckOpen
            itemx.SubItems(1) = "sckOpen"
        Case sckListening
            itemx.SubItems(1) = "sckListening"
        Case sckConnectionPending
            itemx.SubItems(1) = "sckConnectionPending"
        Case sckResolvingHost
            itemx.SubItems(1) = "sckResolvingHost"
        Case sckHostResolved
            itemx.SubItems(1) = "sckHostResolved"
        Case sckConnecting
            itemx.SubItems(1) = "sckConnecting"
        Case sckConnected
            itemx.SubItems(1) = "sckConnected"
        Case sckClosing
            itemx.SubItems(1) = "sckClosing"
        Case sckError
            itemx.SubItems(1) = "sckError"
        Case Else
            itemx.SubItems(1) = "unknown: " & sockClient.State
    End Select

    ' Now set the states for the listening server control as
    ' well as any connected clients
    '
    index = 0
    For i = 2 To ServerIndex + 2
        Set itemx = lstStates.ListItems.Item(i)

        Select Case sockServer(index).State
            Case sckClosed
                itemx.SubItems(1) = "sckClosed"
            Case sckOpen
                itemx.SubItems(1) = "sckOpen"
            Case sckListening
                itemx.SubItems(1) = "sckListening"
            Case sckConnectionPending
                itemx.SubItems(1) = "sckConnectionPending"
```

```
            Case sckResolvingHost
                itemx.SubItems(1) = "sckResolvingHost"
            Case sckHostResolved
                itemx.SubItems(1) = "sckHostResolved"
            Case sckConnecting
                itemx.SubItems(1) = "sckConnecting"
            Case sckConnected
                itemx.SubItems(1) = "sckConnected"
            Case sckClosing
                itemx.SubItems(1) = "sckClosing"
            Case sckError
                itemx.SubItems(1) = "sckError"
            Case Else
                itemx.SubItems(1) = "unknown"
        End Select
        index = index + 1
    Next i
End Sub
```

TCP Server

Now we'll examine the code behind the form. Take a look at the *Form_Load* procedure in Figure 15-5. The first two statements simply set two labels to the host name and IP address of the local machine. These labels are in the Winsock Information group box, which serves the same purpose as the informational box in the UDP example. Next you'll see the initialization of the server control, *sockServer*, to the TCP protocol. Element 0 of the Winsock control array is always the listening socket. After this, the procedure disables the Close Listen button, which is enabled again later when the server starts to listen for clients. The last part of the procedure sets up the *ListView* control, *lstStates*. This control is used to display the current state of every Winsock control currently in use. The code adds entries for the client and server controls so that they are elements 1 and 2, respectively. Any other dynamically loaded Winsock controls will be added after these two. The entry for the server control is named "Local Server." As in the UDP example, the procedure sets a timer to regulate how often the socket states are updated. By default, the timer triggers the update every half second.

From here, let's take a look at the two buttons the server uses. The first is the Listen button, whose function is simple. The handler for the Listen button, *cmdListen*, sets the *LocalPort* property to the value the user entered in the *txtServerPort* text box. The local port is the most important field to a listening socket. This is the port all clients attempt to connect to in order to establish a connection. After setting the *LocalPort* property, all the code needs to do is call the *Listen* method. Once the Listen button's handler puts the *sockServer* control in listening mode, the program waits for the *ConnectionRequest* event to be fired on our *sockServer* control to indicate a client connection. The user can click the other button, Close Listen, to shut down the *sockServer* control. The Close Listen button's handler calls the *Close* method on *sockServer(0),* preventing any further client connections.

The most important event for a TCP server is the *ConnectionRequest* event, which handles incoming client requests. When a client requests a connection, two options exist for handling the request. First, you can use the server socket itself to handle the client. The drawback of this method is that it will close the listening socket and prevent any other connections from being serviced. This method is accomplished by simply calling the *Accept* method on the server control with the *requestID* that is passed into the event handler. The other way to handle a client's connection request is to pass the connection to a separate control. This is what the example SockTCP does. Remember that you have an array of Winsock controls, and element 0 is the listening socket. The first thing to do is search through the array for a control whose state is closed (for example, query the *State* property for the value *sckClosed*). Of course, there are no free controls for the first loop because none are loaded. In this case, the first loop you see is not executed and the variable *freeSock* is still 0, indicating that no free controls were found. The next steps dynamically load a new Winsock control by incrementing the *ServerIndex* counter (the place in the array in which to load the control) and then executing the following statement:

```
Load sockServer(ServerIndex)
```

Now that a new Winsock control is loaded, the procedure can call the *Accept* method with the given request ID. The remaining statements add a new entry in the *lstStates* ListView control so that the program can display the current state of the new Winsock control.

In the case of an already loaded Winsock control whose state is closed, the procedure simply would have reused that control by calling the *Accept* method on it. Continually loading and unloading controls is a bad idea, as it decreases performance. The load and unload processes are expensive. There is also a memory leak when the Winsock control is unloaded that we will discuss in detail later in this chapter.

The remaining server-side functions are straightforward. The *sockServer_Close* event is triggered whenever the client calls the *Close* method on its end. All the server does is close the socket on this side and clear the IP address entry in the ListView control by setting it to "---.---.---.---" and setting the entry's port to -1. The *sockServer_DataArrival* function allocates a buffer to receive the data and then calls the *GetData* method to perform the read. Afterward, the message is added to the *lstMessages* list box. The last server function is the *Error* event handler. Upon an error, the handler displays the text message and closes the control.

TCP Client

Now that you have seen how the server is implemented, let's examine the client. The only initialization performed by the client in the *Form_Load* procedure is setting the protocol of *sockClient* to TCP. Other than the initialization code, three command button handlers belong to the client and several event handlers. The first button is Connect, and its handler is named *cmdConnect_Click*. *LocalPort* is set to 0 because it doesn't matter what the local system assigns as the port number on our machine. *RemoteHost* and *RemotePort* are set according to the values in the *txtServerName* and *txtPort* fields, respectively. That's all the information required to establish a TCP connection to a server. The only task left is to call the *Connect* method. After that, the control's state is in the process of either resolving the name or connecting (the control's state is *sckResolvingHost*, *sckResolved*, or *sckConnecting*). When the connection finally is established, the state changes to *sckConnected* and the *Connect* event is triggered. The next section covers the various states and the transitions among them.

Once the connection is established, the handler *sockClient_Connect* is called. This handler simply enables the Send Data and Disconnect buttons. Additionally, the port number on which the connection is established on the local machine is updated for the Local Client entry in the *lstStates* ListView control. Now you can send and receive data. There are two other event handlers: *sockClient_Close* and *sockClient_Error*. The *sockClient_Close* event handler simply closes the client Winsock control, while the *sockClient_Error* event handler displays a message box with the error description and then closes the control.

The last two pieces to the client are the remaining command buttons: Send Data and Disconnect. The subroutine *cmdSendData_Click* handles the Send Data button. If the Winsock control is connected, the routine calls the *SendData* method with the string in the *txtSendData* text box. Finally, the Disconnect button is handled by *cmdDisconnect_Click*. This handler simply closes the client control, resets a number of buttons to their initial state, and updates the Local Client entry in the *lstStates* ListView control.

Obtaining Winsock Information

The last part of the TCP example is the Winsock information section. We have already explained this a bit, but we'll briefly present it here for the sake of clarity. As with the UDP example, a timer triggers an update on the current socket states of all loaded Winsock controls. The default refresh rate is set to 500 milliseconds. Upon load, two entries are added to the *lstStates* Listview control. The first is the Local Client label that corresponds to the client Winsock control, *sockClient*. The second entry is Local Server, which refers to the listening socket. Whenever a new client connection is established, a new Winsock control is dynamically loaded and a new entry is added to the *sckStates* control; the name of the entry is the client's IP address. When a client disconnects, the entry is set to the default state with the IP address "---.---.---.---" and port number equal to -1. Of course, if another client connects, it reuses any unused controls in the server array. The local machine's IP address and host name are also displayed.

Running the TCP Example

Again, running the TCP example is a straightforward process. Start three instances of the application, each on a separate machine. With TCP, it doesn't matter if any of the machines are multihomed because the routing table decides which interface is more appropriate for any given TCP connection. On one of the TCP examples, start the listening socket by clicking the Listen button. You'll notice that the Local Server entry in the State Information

ListView control changes from *sckClosed* to *sckListening* and the port number is listed as 5150. The server is now ready to accept client connections.

On one of the clients, set the Server Name field to the name of the machine running the first instance of the application (the listening server) and then click the Connect button. On the client application, the Local Client entry in the State Information list is now in the *sckConnected* state and the local port on which the connection was made is updated to a non-negative number. Additionally, on the server side, an entry is added to the State Information list whose name is the IP address of the client that just connected. The new entry's state is *sckConnected* and also contains the port number on the server side on which the connection was established. Now you can type text into the Message text field on the client and click the Send Data button a few times. You will see the messages appearing in the Messages list box on the server side. Next disconnect the client by clicking the Disconnect button. On the client side, the Local Client entry in the State Information list is set back to *sckClosed* and the port number value to -1. For the server, the entry corresponding to the client is not removed; it is simply marked as unused with the name set to a dashed IP address, the state to *sckClosed*, and the port to -1.

On the third machine, enter the name of the listening server in the Server Name text box and make a client connection. The results are similar to those for the first client except that the server uses the same Winsock control to handle this client as it did the first. If a Winsock control is in the closed state, it can be used to accept any incoming connection. The final step you might want to try is using the client on the server application to make a connection locally. After you make the connection, a new entry is added to the Socket Information list, as in the earlier examples. The only difference is that the IP listed is the same as the IP address of the server. Play with the clients and server a bit to get a feel for how they interact and what results are triggered by each command.

TCP States

Using the Winsock control with the TCP protocol is much more complicated than using UDP sockets because many more socket states are possible. Figure 15-6 is a state diagram for a TCP socket. The default start state is *sckClosed*. The transitions are straightforward and don't require explanation except for the *sckClosing* state. Because of the TCP half-close, there are two transition paths from this state for the *SendData* method. Once one side of the TCP connection issues a *Close* method, that side can't send any more data. The other side of the connection receives the *Close* event and enters the *sckClosing* state but can still send data. This is why there are two paths out of *sckClosing* for the *SendData* method. If the side that issued the *Close* tries to call *SendData*, an error is generated and the state moves to *sckError*. The side that receives the *Close* event can freely send data and receive any remaining data.

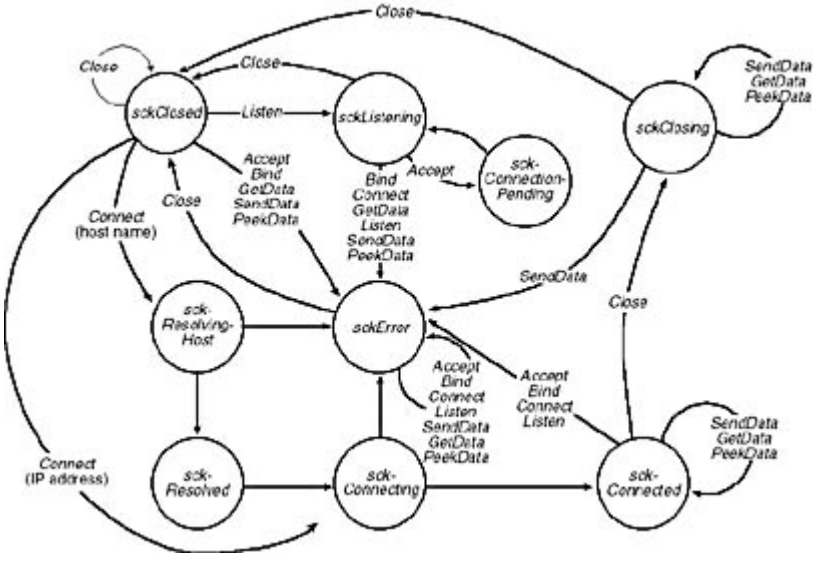

Figure 15-6. *TCP state diagram*

Limitations

The Winsock control is clearly useful and easy to use; unfortunately, a few bugs make the control unusable for mission-critical applications. The bugs discussed in this section apply to the latest version of the control for Visual Basic 5.0, which is the updated control from Service Pack 2.

The first bug is relatively minor and deals with dynamically loading and unloading the control. A memory leak is incurred when unloading a previously loaded control. This is why we don't load and unload the controls as clients connect and disconnect in our server example. Once the control is loaded in memory, we leave it for possible use by other clients.

The second bug involves closing a socket connection before all data queued is sent on the wire. In some cases, calling the *Close* method after the *SendData* event (when *Close* is processed before *SendData*) causes data to be lost, at least from the receiver's point of view. You can get around this problem by catching the *SendComplete* event (which is triggered when *SendData* has finished putting the data on the wire). Alternatively, you could arrange the send/receive transactions so that the receiver issues the *Close* command first, when it has received all the data expected. This would then trigger the *Close* event on the sender, which would then signal that all the data sent has been received, and that it's now OK to shut down the connection completely.

The last and most severe bug is the dropping of data when a large buffer is submitted for transfer. If a large enough block of data is queued up for network transmission, the control's internal buffers get messed up and some data is dropped. Unfortunately, there is no completely perfect workaround for this problem. The best method is to submit data in chunks less than 1000 bytes. Once a buffer is submitted, wait for the *SendComplete* event to fire before submitting the next buffer. This is a pain, but it's still the best way to make the control as reliable as possible.

The latest Winsock control shipping in Visual Basic 6.0 has fixed the bugs above except for the second one. If you issue a *Close* command after calling *SendData*, the socket closes immediately without sending all the data. While it would have been wonderful to have all the bugs fixed, the remaining problem is perhaps the least severe of the three and the easiest to work around.

Common Errors

As you saw in Chapters 6 through 14, an application can encounter quite a few Winsock errors. We won't go into all of them here. However, in the following two sections we will discuss the errors that are most commonly encountered by applications using the Winsock control: "Local address in use" and "Invalid operation at current state."

Local Address In Use

The "Local address in use" error occurs when you bind to a local port, either through the *Bind* method or the *Connect* method, but find that the port is already in use. This is most often encountered in the TCP server that always binds to a specific port so clients can locate the service. If a socket is not properly closed before an application using that socket exits, the socket goes into the TIME_WAIT state for a short period of time to ensure that all data has been sent or received on that port. If an attempt is made to bind to that port, the "Local address in use" error is generated. A common mistake on the client side also results in this error. If the *LocalPort* property is set to 0 and a connection is established, *LocalPort* is updated to the port number on which the client connection was made locally. If you plan to reuse the same control to make a subsequent connection, be sure you reset the *LocalPort* value to 0. Otherwise, if the previous connection was not properly shut down, you might run into this error.

Invalid Operation at Current State Run-Time Error

The "Invalid operation at current state" error is probably the most frequently seen error. It occurs when a Winsock control method is called but the current state of the control prohibits that action. Take a look at Figures 15-3 and 15-6 for the state diagrams for UDP and TCP sockets. To write robust code, always check the socket's state before calling a method.

Winsock errors will be generated through the *Error* event. These are the same errors as the errors from straight Winsock programming. For a more detailed description of Winsock errors, refer to Chapter 7, which covers the most common errors encountered, or consult Appendix C, which lists all possible Winsock error codes.

The Windows CE Winsock Control

The Visual Basic Toolkit for Windows CE (VBCE) contains a Winsock control that provides many of the same capabilities offered by the "regular" Visual Basic Winsock control. The major difference is that while UDP is unsupported, the Windows CE Winsock control offers the IrDA protocol. Also, some minor differences between the two controls require some programmatic changes from what you have seen with the non_Windows CE Winsock control.

As you read in Chapter 7, Windows CE does not offer the asynchronous Winsock model. The Windows CE Winsock control is no exception. The main difference in programming is that the *Connect* method is blocking. There is no *Connect* event. Once you attempt a connection by calling *Connect*, the call will block until a connection is made or an error is returned.

Additionally, the VBCE 1.0 does not support control arrays, which means you will have to change your server design from that presented in Figure 15-5. As a result, the only way to handle multiple connections is to place a number of Windows CE Winsock controls onto the form. Realistically, this limits the maximum number of concurrent client connections that you can handle because this solution does not scale at all.

Finally, the *ConnectionRequest* event doesn't have a *RequestID* parameter, which might seem a bit strange. The end result is that you must call the *Accept* method on the control to which the connection will be handed off. The connection request that triggers the *ConnectionRequest* event is handled by the control that receives the connection request.

Windows CE Winsock Example

In this section, we'll briefly introduce a sample application using the Windows CE Winsock control. The same principles apply to the Windows CE control as to the desktop Winsock control except for the differences noted above. Figure 15-7 shows the code behind the Windows CE Winsock control.

Figure 15-7. *The Windows CE Winsock example*

```
Option Explicit

' This global variable is used to retain the current value
' of the radio buttons.  0 corresponds to TCP, while 2 means
' IrDA (infrared).  Note that UDP is not supported by the
' control currently.
Public SocketType

Private Sub cmdCloseListen_Click()
' Close the listening socket, and set the other buttons
' back to the start state
    WinSock1.Close

    cmdConnect.Enabled = True
    cmdListen.Enabled = True
    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
End Sub

Private Sub cmdConnect_Click()
' Check which type of socket type was chosen, and initiate
' the given connection
```

```
    If SocketType = 0 Then
        ' Set the protocol and the remote host name and port
        '
        WinSock1.Protocol = 0
        WinSock1.RemoteHost = txtServerName.Text
        WinSock1.RemotePort = CInt(txtPort.Text)
        WinSock1.LocalPort = 0

        WinSock1.Connect
    ElseIf SocketType = 2 Then
        ' Set the protocol to IrDA, and set the service name
        '
        WinSock1.Protocol = 2
        'WinSock1.LocalPort = 0
        'WinSock1.ServiceName = txtServerName.Text
        WinSock1.RemoteHost = txtServerName.Text

        WinSock1.Connect
    End If
    ' Make sure the connection was successful; if so,
    ' enable/disable some commands
    '
    MsgBox WinSock1.State
    If (WinSock1.State = 7) Then
        cmdConnect.Enabled = False
        cmdListen.Enabled = False
        cmdDisconnect.Enabled = True
        cmdSendData.Enabled = True
    Else
        MsgBox "Connect failed"
        WinSock1.Close
    End If
End Sub

Private Sub cmdDisconnect_Click()
' Close the current client connection, and reset the
' buttons to the start state
    WinSock1.Close

    cmdConnect.Enabled = True
    cmdListen.Enabled = True
    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
End Sub

Private Sub cmdListen_Click()
' Set the socket to listening mode for the given protocol
' type
'
    If SocketType = 0 Then
        WinSock1.Protocol = 0
```

```
        WinSock1.LocalPort = CInt(txtLocalPort.Text)
        WinSock1.Listen
    ElseIf SocketType = 2 Then
        WinSock1.Protocol = 2
        WinSock1.ServiceName = txtServerName.Text
        WinSock1.Listen
    End If
    ' If we're not in listening mode now, something
    ' went wrong
    '
    If (WinSock1.State = 2) Then
        cmdConnect.Enabled = False
        cmdListen.Enabled = False
        cmdCloseListen.Enabled = True
    Else
        MsgBox "Unable to listen!"
    End If
End Sub

Private Sub cmdSendData_Click()
' Send the data in the box on the current connection
'
    WinSock1.SendData txtSendData.Text
End Sub

Private Sub Form_Load()
' Set the initial values for the buttons, the timer, etc.
'
    optTCP.Value = True
    SocketType = 0

    Timer1.Interval = 750
    Timer1.Enabled = True

    cmdConnect.Enabled = True
    cmdListen.Enabled = True

    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False

    lblLocalIP.Caption = WinSock1.LocalIP
End Sub

Private Sub optIRDA_Click()
' Set the socket type to IrDA
'
    optIRDA.Value = True
    SocketType = 2
End Sub

Private Sub optTCP_Click()
' Set the socket type to TCP
```

```
'
    optTCP.Value = True
    SocketType = 0
    cmdConnect.Caption = "Connect"
End Sub

Private Sub Timer1_Timer()
' This is the event that gets called each time the
' timer expires. Update the socket state label.
'
    Select Case WinSock1.State
        Case 0
            lblState.Caption = "sckClosed"
        Case 1
            lblState.Caption = "sckOpen"
        Case 2
            lblState.Caption = "sckListening"
        Case 3
            lblState.Caption = "sckConnectionPending"
        Case 4
            lblState.Caption = "sckResolvingHost"
        Case 5
            lblState.Caption = "sckHostResolved"
        Case 6
            lblState.Caption = "sckConnecting"
        Case 7
            lblState.Caption = "sckConnected"
        Case 8
            lblState.Caption = "sckClosing"
        Case 9
            lblState.Caption = "sckError"
    End Select
End Sub

Private Sub WinSock1_Close()
' The other side initiated a close, so we'll close our end.
' Reset the buttons to their initial state.
'
    WinSock1.Close

    cmdConnect.Enabled = True
    cmdListen.Enabled = True

    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
End Sub

Private Sub WinSock1_ConnectionRequest()
' We got a client connection; accept it on the listening
' socket
'
    WinSock1.Accept
```

```
End Sub

Private Sub WinSock1_DataArrival(ByVal bytesTotal)
' This is the event for data arrival.  Get the data, and
' add it to the list box.
'
    Dim rdata

    WinSock1.GetData rdata
    List1.AddItem rdata
End Sub

Private Sub WinSock1_Error(ByVal number, ByVal description)
' An error occurred; display the message, and close the socket
'
    MsgBox description
    Call WinSock1_Close
End Sub
```

We won't go into the specifics of the sample code in Figure 15-7 because it is similar to the SockTCP example in Figure 15-5. The only differences between the two are the known limitations mentioned in the previous section. One thing you will notice is that the Windows CE Winsock control is a bare-bones control. That is, it isn't as well polished as the desktop version. The type libraries aren't fully implemented: you must differentiate the protocol type with a simple integer as opposed to an enumerated type. In addition, there is the problem with the socket state enumerated type mentioned in the next section, "Known Problems."

Handling infrared connections is not that different from handling TCP connections. The one exception occurs when a listening socket is established over the infrared port. An infrared server is known by its *service name,* which is discussed in detail in the IrDA addressing section of Chapter 6. The Windows CE Winsock control has an additional property named *ServiceName*. You set this property to the text string that clients attempt to connect to. For example, the following code snippet puts the Windows CE Winsock control named *CeWinsock* into listening mode under the name "MyServer."

```
CeWinsock.Protocol = 2              ' Protocol 2 is IrSock
CeWinsock.ServiceName = "MyServer"
CeWinsock.Listen
```

There are no other requirements for publishing a service under infrared sockets. You need to specify only the service name.

Known Problems

The one rather strange problem we encountered with the VBCE Winsock control is the use of the enumerated values for the Winsock states. For some odd reason, these values are defined in the development environment, but on the remote device the following error message pops up every time you reference an *sck* enumerated value in your code: "An error was encountered while running this program." If you replace the enumerations with their constant equivalents, these errors go away. This has been marked as a bug and will be corrected in a future release of the toolkit.

Conclusion

The Visual Basic Winsock control is useful for simple, noncritical applications that require network communication. A few problems with the Visual Basic 5.0 version of the control make successfully programming the control difficult, but most of the major problems have been corrected in the latest version of Visual Basic. The control offers the ability to add simple network communication to a Visual Basic application with relatively little effort. Of course, the control is limited in its overall abilities, and applications that require a great deal of interaction with Winsock should consider manually importing the necessary functions and constants from the Winsock DLL. As we mentioned earlier, we have provided Winsock Visual Basic examples throughout Part II that do in fact import Winsock functions from Ws2_32.dll. For examples, see the SimpleTCP and SimpleUDP applications under the Chapter07\VB directory and their Winsock.bas file.

Part III

Remote Access Service (RAS)

So far, this book has described all the networking API functions available in Microsoft Windows that allow you to develop applications capable of communicating with other applications over a network. For the most part, our discussions have focused on the application and presentation layers of the seven-layer OSI model. We did not describe the details of a specific network protocol, and most of our discussions centered on using the functions from a protocol-independent standpoint.

The final section of this book describes an important service named Remote Access Service (RAS) that allows users to connect their computer from a remote location to a local computer network. Once connected, you can use the network functions described throughout this book as though your computer were directly connected to a remote network.

Chapter 16

Remote Access Service Client

All Microsoft Windows platforms feature a Remote Access Service (RAS) client, which allows you to connect your computer from a remote location to another computer featuring a remote access server component. Typically, a RAS client will do this by using a modem that connects to a telephone line and calls the remote computer by dialing a telephone number. Because of this, the RAS client is sometimes referred to as a dial-up networking (DUN) client.

On the server side, you must have a service awaiting your DUN connection. A RAS client is capable of establishing a communication link with several types of remote access servers. RAS does this by using industry standard framing protocols, such as the following:

- Point-to-Point Protocol (PPP) Can transmit the IP, IPX, and NetBEUI communication protocols
- Serial Line Internet Protocol (SLIP) Can transmit only the IP communication protocol
- Asynchronous NetBEUI (Microsoft Windows NT 3.1, Microsoft Windows for Workgroups 3.11) Can transmit only the NetBEUI communication protocol

The framing protocols describe how data is transmitted over a RAS communication link and dictate which network communication protocols (such as TCP/IP or IPX) can communicate over the RAS link. If a RAS server supports one of the framing protocols defined in the previous list, a RAS client can establish a connection. Windows 95, Windows 98, Windows 2000, and Windows NT feature a RAS server component capable of supporting all the framing protocols listed.

Once a connection is established between a RAS client and server, network protocol stacks (depending on the framing protocol used) can communicate over the RAS connection to the remote computer as if the computers were connected over a LAN. Of course, the data communication rate of many modems today is significantly slower than a direct LAN connection.

When a RAS server accepts a dial-up connection, it first establishes communication with your client by negotiating one of the framing protocols in the previous list. Once the framing protocol is established, the RAS server attempts to authenticate the user that is making the connection. The RAS API functions described in this chapter allow a RAS client to specify a user name, a password, and domain logon credentials to the RAS server. When a Windows 2000 or a Windows NT RAS server receives this information, it validates these logon credentials using Windows NT domain security access control. Note that the RAS server does not log your client on to a Windows NT domain; instead, it uses the client credentials to verify that a user is allowed to make a RAS connection. The RAS connection process is not the same as the Windows NT domain logon process. After a RAS connection is successfully established, your computer can log on to a Windows NT domain. This chapter doesn't describe the process of logging on to a domain. On Windows 95 and Windows 98, RAS can automatically log a machine on to a domain after a RAS connection is authenticated through options available in a phonebook entry, as we will discuss later in this chapter.

RAS relies on the Telephony Application Programming Interface (TAPI) to set up and control telephone communication devices such as modems on your computer. TAPI controls the hardware settings of these dialing devices. When you set up a RAS connection using a modem, TAPI turns on the modem and sends dialing information from RAS to the modem. As a result, RAS views modems as simple TAPI modem ports that are capable of dialing and making a phone connection to a remote server. As you will see later in this chapter, some of the RAS API functions refer to TAPI modem ports when you set up RAS connection information.

This chapter will explain how you can programmatically use RAS to establish remote network communication. We will begin by describing the header and library files you need to build your application. Next we will describe the basics of dialing—how you actually establish a remote connection. We'll then describe how you can set up RAS phonebook entries to define detailed communication properties of a RAS connection. Once we've explained the basics of setting up communication, we'll show you how to manage established connections.

Compiling and Linking

When you develop a RAS application, you need to include the following header and library files to build your application:

- Ras.h Contains the function prototypes and data structures used by RAS API functions
- Raserror.h Contains predefined error codes used in RAS API functions when they fail
- Rasapi32.lib Library of all RAS API functions

Raserror.h lists quite a few predefined error codes. In this file, you will notice an error description string associated with each error code used in RAS. RAS features a useful function named *RasGetErrorString* that allows you to programmatically retrieve the error strings associated with specific RAS error codes. *RasGetErrorString* is defined as

```
DWORD RasGetErrorString(
    UINT uErrorValue,
    LPTSTR lpszErrorString,
    DWORD cBufSize
);
```

The *uErrorValue* parameter receives a specific RAS error code returned from a RAS function. The *lpszErrorString* parameter is an application-supplied buffer that will receive the error string associated with the error code in *uErrorValue*. You should make your buffer large enough to hold an error string; otherwise, this function will fail with error *ERROR_INSUFFICIENT_BUFFER*. We recommend setting your buffer size to at least 256 characters, which should accommodate any RAS error string available today. The final parameter, *cBufSize*, is the size of the buffer you supplied to *lpszErrorString*.

Data Structures and Platform Compatibility Issues

When you compile and build your application, you will find that some of the data structures used by the RAS functions have extra data fields included or excluded, based on the value of the define *WINVER*. However, the Windows CE SDK does not define *WINVER*, so this information does not apply to Windows CE. RAS data structures also have a *dwSize* field that you must set to the byte size of the RAS structure you are using. This affects the behavior of the RAS functions that use these structures because they are targeted for a specific platform. The *WINVER* rules apply to the Windows 95, Windows 98, Windows 2000, and Windows NT platforms.

- *WINVER* = 0x400 Indicates that your RAS application is targeted for Windows 95, Windows 98, or Windows NT 4 with no service pack
- *WINVER* = 0x401 Indicates that your RAS application is targeted for Windows NT 4 with any service pack
- *WINVER* = 0x500 Indicates that your RAS application is targeted for Windows 2000

RAS does not lend itself well to having a single executable that can run on all platforms. Through careful programming, it is possible to support all platforms (except Windows CE, of course) using a single executable. However, we highly recommend targeting a specific platform when building your RAS applications.

DUN 1.3 Upgrade and Windows 95

A number of Windows 95 releases occurred between the original release and the OSR 2 release. The OSR 2 release was not a retail product; instead, it was available only for original equipment manufacturers (OEMs) to install on a new computer. Each release features a different flavor of the RAS API functionality. Because of this, we recommend that you install the latest RAS upgrade package—DUN 1.3—to raise RAS to the current functionality level of newer platforms. You can obtain the DUN 1.3 upgrade for Windows 95 at *http://www.microsoft.com/support*. This chapter assumes that you have installed at least the DUN 1.3 upgrade; we will not address RAS issues from earlier DUN versions.

RasDial

When a RAS client application is ready to make a connection to a remote computer, it must call the *RasDial* function. *RasDial* is quite complex, offering many call parameters that are used for dialing, authenticating, and establishing a remote connection to a RAS server. *RasDial* is defined as

```
DWORD RasDial(
    LPRASDIALEXTENSIONS lpRasDialExtensions,
    LPCTSTR lpszPhonebook,
    LPRASDIALPARAMS lpRasDialParams,
    DWORD dwNotifierType,
    LPVOID lpvNotifier,
    LPHRASCONN lphRasConn
);
```

The *lpRasDialExtensions* parameter is an optional pointer to a *RASDIALEXTENSIONS* structure that causes your application to enable extended features for *RasDial*. On Windows 95, Windows 98, and Windows CE, this parameter is ignored and should be set to *NULL*. The *RASDIALEXTENSIONS* structure is defined as

```
typedef struct tagRASDIALEXTENSIONS {
   DWORD      dwSize;
   DWORD      dwfOptions;
   HWND       hwndParent;
   ULONG_PTR  reserved;
#if (WINVER >= 0x500)
   ULONG_PTR  reserved1;
   RASEAPINFO RasEapInfo;
#endif
} RASDIALEXTENSIONS;
```

Notice that this structure is sized differently during compilation based on the value of *WINVER*, as described earlier. The above fields are described as follows:

- *dwSize* Should be set to the size (in bytes) of the *RASDIALEXTENSIONS* structure.
- *dwfOptions* Allows you to set bit flags for using *RasDial* extensions. Table 16-1 describes these flags.

Table 16-1. RasDial *bit flags of the extension features*

Flag	*Description*
RDEOPT_UsePrefixSuffix	Makes *RasDial* use the prefix and suffix associated with the specified dialing device.
RDEOPT_PausedStates	Allows *RasDial* to enter a paused operating state so that users can retry logons, change passwords, and set callback numbers.
RDEOPT_IgnoreModemSpeaker	Makes *RasDial* ignore the modem speaker settings in the RAS phonebook.
RDEOPT_SetModemSpeaker	Turns on the modem speaker if *RDEOPT_IgnoreModemSpeaker* is set.
RDEOPT_IgnoreSoftwareCompression	Makes *RasDial* ignore software compression settings.
RDEOPT_SetSoftwareCompression	Turns on software compression if *RDEOPT_IgnoreSoftwareCompression* is set.
RDEOPT_PauseOnScript	Used internally by *RasDialDlg*. You should not set this flag.

- *hwndParent* Is not used and should be set to *NULL*.
- *reserved* Is not used and should be set to 0.
- *reserved1* Is reserved for future use on Windows 2000 and should be set to 0.
- *RasEapInfo* On Windows 2000, allows you to specify Extensible Authentication Protocol (EAP) information. The details of EAP are beyond the scope of this book.

The *lpszPhonebook* parameter of *RasDial* identifies the path to a phonebook file on Windows 2000 and Windows NT. This parameter must be *NULL* on Windows 95, Windows 98, and Windows CE because the phonebook is stored in the system Registry. A phonebook is a collection of RAS dialing properties that define how to set up a RAS connection. However, you are not required to use a phonebook to make a RAS connection. *RasDial* features enough dialing parameters to allow you to set up a basic connection. We will discuss the details of a RAS phonebook later in this chapter.

The *RASDIALPARAMS* structure pointer *lpRasDialParams* defines dialing and user authentication parameters that the *RasDial* function uses to establish a remote connection. It's defined as

```
typedef struct _RASDIALPARAMS {
    DWORD dwSize;
    TCHAR szEntryName[RAS_MaxEntryName + 1];
    TCHAR szPhoneNumber[RAS_MaxPhoneNumber + 1];
    TCHAR szCallbackNumber[RAS_MaxCallbackNumber + 1];
    TCHAR szUserName[UNLEN + 1];
    TCHAR szPassword[PWLEN + 1];
    TCHAR szDomain[DNLEN + 1] ;
#if (WINVER >= 0x401)
    DWORD dwSubEntry;
    DWORD dwCallbackId;
#endif
} RASDIALPARAMS;
```

The fields of *RASDIALPARAMS* are described as follows:

- *dwSize* Should be set to the size (in bytes) of a *RASDIALPARAMS* structure. This allows RAS to internally

determine what *WINVER* version you compiled with.

- *szEntryName* A string that allows you to identify a phonebook entry contained in the phonebook file listed in the *lpszPhonebook* parameter of *RasDial*. This is an important parameter because phonebook entries enable you to fine-tune RAS connection properties such as selecting a modem or selecting a framing protocol. However, specifying a phonebook entry to use *RasDial* is optional. If this field is an empty string (""), *RasDial* will select the first available modem installed on your system and will rely on the next parameter, *szPhoneNumber*, to dial a connection.

- *szPhoneNumber* A string representing a phone number that overrides the number contained in the phonebook entry specified in the *szEntryName* field.

- *szCallbackNumber* Allows you to specify a phone number the RAS server can call you back on. If the RAS server permits you to have a callback number, the server will terminate your original connection and call back your client using the callback number you specified. This is a nice feature because it lets your server know where a user is connecting from.

- *szUserName* A string that identifies a logon name used to authenticate a user on a RAS server.

- *szPassword* A string that identifies the password used to authenticate a user on a RAS server.

- *szDomain* Identifies the Windows 2000 or Windows NT domain where the user account is located.

- *dwSubEntry* Optionally allows you to specify the initial phonebook subentry to dial for a RAS multilink connection. (We'll describe RAS phonebook subentries and multilink connections in "Multilink Phonebook Subentries.")

- *dwCallbackId* Allows you to pass an application-defined value to a *RasDialFunc2* callback function (which we'll also describe later). If you're not using a *RasDialFunc2* callback function, this field is not used.

The next *RasDial* parameters, *dwNotifierType* and *lpvNotifier*, determine the operating mode of *RasDial* (whether it can be called synchronously or asynchronously). The final *RasDial* parameter, *lphRasConn*, is a pointer to a RAS connection handle of type *HRASCONN*. Before you call *RasDial*, you must set this parameter to *NULL*. If *RasDial* completes successfully, a reference handle to the RAS connection is returned.

Let's begin by demonstrating how to call *RasDial*. As we mentioned, *RasDial* can execute in two operating modes: *synchronous* and *asynchronous*. In synchronous mode, *RasDial* blocks until it either completes a connection or fails to do so. In asynchronous mode, *RasDial* completes a connection immediately, allowing your application to perform other actions while connecting.

Synchronous Mode

If the *lpvNotifier* parameter of *RasDial* is set to *NULL*, *RasDial* will operate synchronously. When the *lpvNotifier* parameter is *NULL*, the *dwNotifierType* parameter is ignored. Calling *RasDial* synchronously is the easiest way to use this function; however, you won't be able to monitor the connection as you can in asynchronous mode, which we will describe in a moment. Figure 16-1 demonstrates how to call *RasDial* synchronously. Notice that this code listing does not specify a phonebook or a phonebook entry. Instead, it demonstrates how simple forming a RAS connection is.

Figure 16-1. *Calling* RasDial *synchronously*

```
RASDIALPARAMS RasDialParams;
HRASCONN hRasConn;
DWORD Ret;

// Always set the size of the RASDIALPARAMS structure

RasDialParams.dwSize = sizeof(RASDIALPARAMS);
hRasConn = NULL;

// Setting this field to an empty string will allow
// RasDial to use default dialing properties

lstrcpy(RasDialParams.szEntryName, "");

lstrcpy(RasDialParams.szPhoneNumber, "867-5309");
lstrcpy(RasDialParams.szUserName, "jenny");
lstrcpy(RasDialParams.szPassword, "mypassword");
lstrcpy(RasDialParams.szDomain, "mydomain");

// Call RasDial synchronously (the fifth parameter
// is set to NULL)
Ret = RasDial(NULL, NULL, &RasDialParams, 0, NULL, &hRasConn);

if (Ret != 0)
{
    printf("RasDial failed: Error = %d\n", Ret);
}
```

Asynchronous Mode

Calling *RasDial* asynchronously is a lot more complicated than calling this function in synchronous mode. If the *lpvNotifier* parameter of *RasDial* is not set to *NULL*, *RasDial* will operate asynchronously—meaning the call returns immediately but the connection proceeds. Calling *RasDial* asynchronously is the preferred method for making a RAS connection because you can monitor the connection's progress. The *lpvNotifier* parameter can be either a pointer to a function that is called when a connection activity occurs in *RasDial* or a window handle that receives progress notification via Windows messages. The *dwNotifierType* parameter of *RasDial* determines the type of function or window handle that is passed into *lpvNotifier*. Table 16-2 describes the values that you can specify in *dwNotifierType*.

Table 16-2. RasDial *asynchronous notification methods*

Notifier Type	*Meaning*
0	The *lpvNotifier* parameter causes *RasDial* to use the *RasDialFunc* function pointer to manage connection events.
1	The *lpvNotifier* parameter causes *RasDial* to use the *RasDialFunc1* function pointer to manage connection events.
2	The *lpvNotifier* parameter causes *RasDial* to use the *RasDialFunc2* function pointer to manage connection events.
0xFFFFFFFF	The *lpvNotifier* parameter makes *RasDial* send a window message during connection events.

Table 16-2 shows the three function prototypes that you can supply to *RasDial* in the *lpvNotifier* parameter for receiving callback notification of connection events: *RasDialFunc*, *RasDialFunc1*, and *RasDialFunc2*. The first one, *RasDialFunc*, is prototyped as

```
VOID WINAPI RasDialFunc(
    UINT unMsg,
    RASCONNSTATE rasconnstate,
    DWORD dwError
);
```

The *unMsg* parameter receives the type of event that has occurred. Currently this event can be only *WM_RASDIALEVENT*, which means that this parameter is not useful. The *rasconnstate* parameter receives the connection activity that the *RasDial* function is about to start. Table 16-3 defines the possible connection activities. The *dwError* parameter receives a RAS error code if one of the connection activities experiences failure.

Table 16-3 shows three operating states associated with connection activities in an asynchronous *RasDial* call: running, paused, and terminal. The running state indicates that the *RasDial* call is still in progress, and each running-state activity offers progress status information.

The paused state indicates that *RasDial* needs more information to establish the connection. By default, the paused state is disabled. You can enable this notification process by setting the *RDEOPT_PausedStates* flag in the *RASDIALEXTENSIONS* structure that we mentioned earlier. When a paused state activity occurs, it indicates one of the conditions listed below.

- The user needs to supply new logon credentials because the authentication failed.
- The user needs to provide a new password because his or her password has expired.
- The user needs to provide a callback number.

Table 16-3. *RAS connection activities*

Activity	*State*	*Description*
RASCS_OpenPort	Running	A communication port is about to be opened.
RASCS_PortOpened	Running	The communication port is open.
RASCS_ConnectDevice	Running	A device is about to be connected.
RASCS_ DeviceConnected	Running	The device has successfully connected.
RASCS_ AllDevicesConnected	Running	A physical link has been established.
RASCS_ Authenticate	Running	The RAS authentication process has started.
RASCS_ AuthNotify	Running	An authentication event has occurred.
RASCS_ AuthRetry	Running	The client has requested another authentication attempt.
RASCS_ AuthCallback	Running	The server has requested a callback number.
RASCS_ AuthChangePassword	Running	The client has requested to change the password on the RAS account.
RASCS_ AuthProject	Running	The protocol projection is starting.
RASCS_ AuthLinkSpeed	Running	The link speed is being calculated.
RASCS_ AuthAck	Running	An authentication request is being acknowledged.

RASCS_ ReAuthenticate	Running	The authentication process after a callback is starting.
RASCS_ Authenticated	Running	The client has successfully completed the authentication.
RASCS_ PrepareForCallback	Running	The line is about to disconnect to prepare for a callback.
RASCS_ WaitForModemReset	Running	The client is waiting for the modem to reset before preparing for a callback.
RASCS_ WaitForCallback	Running	The client is waiting for an incoming call from the server.
RASCS_ Projected	Running	The protocol projection is complete.
RASCS_ StartAuthentication	Running	User authentication is being started or retried. (This applies to Windows 95 and Windows 98 only.)
RASCS_ CallbackComplete	Running	The client has been called back. (This applies to Windows 95 and Windows 98 only.)
RASCS_ LogonNetwork	Running	The client is logging on to a remote network. (This applies to Windows 95 and Windows 98 only.)
RASCS_ SubEntryConnected	Running	A subentry of a multilink phonebook entry has connected. The *dwSubEntry* parameter of *RasDialFunc2* will contain an index of the subentry connected.
RASCS_ SubEntryDisconnected	Running	A subentry of a multilink phonebook entry has disconnected. The *dwSubEntry* parameter of *RasDialFunc2* will contain an index of the subentry disconnected.
RASCS_ RetryAuthentication	Paused	*RasDial* is awaiting new user credentials.
RASCS_ CallbackSetByCaller	Paused	*RasDial* is awaiting a callback number from the client.
RASCS_ PasswordExpired	Paused	*RasDial* expects the user to supply a new password.
RASCS_InvokeEapUI	Paused	On Windows 2000, *RasDial* is awaiting a custom user interface to obtain EAP information.
RASCS_ Connected	Terminal	The RAS connection succeeded and is active.
RASCS_ Disconnected	Terminal	The RAS connection failed or is inactive.

These activities pertain to information in the *RASDIALPARAMS* structure described earlier in this chapter. When a paused state activity occurs, *RasDial* will notify your callback function (or window procedure). If the paused state is disabled, RAS will send an error to your notification function and *RasDial* will fail. If enabled, the *RasDial* function will be in a paused state that allows your application to supply the necessary information through a *RASDIALPARAMS* structure. When *RasDial* is paused, you can resume by calling it again with the original call's connection handle (*lphRasConn*) and notification function (*lpvNotifier*), or you can simply end the paused operation by calling *RasHangUp* (described later in this chapter). If you resume the paused connection, you will have to supply the necessary user input via the *RASDIALPARAMS* structure passed to the resumed *RasDial* call.

> NOTE
>
> Do not resume the paused state by calling *RasDial* directly from a notification handler function such as *RasDialFunc*. *RasDial* is not designed to handle this situation, so you should resume *RasDial* directly from your application thread.

The final state—terminal—indicates that the *RasDial* connection has either succeeded or failed. It can also indicate that the *RasHangUp* function closed the connection.

Now that you have a basic understanding of how you can monitor the connection of an asynchronous *RasDial* call, we'll demonstrate how to set up a simple program that calls *RasDial* asynchronously. Figure 16-2 shows this procedure. You'll also find an asynchronous *RasDial* example on the companion CD.

Figure 16-2 *Calling* RasDial *asynchronously*

```
void main(void)
{
    DWORD Ret;
    RASDIALPARAMS RasDialParams;
    HRASCONN hRasConn;

    // Fill in the RASDIALPARAMS structure with call parameters
    // as was done in the synchronous example
    ...

    if ((Ret = RasDial(NULL, NULL, &RasDialParams, 0,
        &RasDialFunc, &hRasConn)) != 0)
    {
        printf("RasDial failed with error %d\n", Ret);
        return;
    }

    // Perform other tasks while RasDial is processing
    ...
}

// Callback function RasDialFunc()
void WINAPI RasDialFunc(UINT unMsg, RASCONNSTATE rasconnstate,
    DWORD dwError)
{
    char szRasString[256]; // Buffer for error string

    if (dwError)
    {
        RasGetErrorString((UINT)dwError, szRasString, 256);
        printf("Error: %d - %s\n",dwError, szRasString);
        return;
    }

    // Map each of the states of RasDial, and display on the
    // screen the next state that RasDial is entering

    switch (rasconnstate)
    {
        case RASCS_ConnectDevice:
            printf ("Connecting device...\n");
            break;
        case RASCS_DeviceConnected:
            printf ("Device connected.\n");
            break;
```

```
        // Add other connection activities here
        ...

        default:
            printf ("Unmonitored RAS activity.\n");
            break;
    }
}
```

Table 16-2 also mentioned two other callback notification functions: *RasDialFunc1* and *RasDialFunc2*. These functions are prototyped as

```
VOID WINAPI RasDialFunc1(
    HRASCONN hrasconn,
    UINT unMsg,
    RASCONNSTATE rascs,
    DWORD dwError,
    DWORD dwExtendedError
);

DWORD WINAPI RasDialFunc2(
    DWORD dwCallbackId,
    DWORD dwSubEntry,
    HRASCONN hrasconn,
    UINT unMsg,
    RASCONNSTATE rascs,
    DWORD dwError,
    DWORD dwExtendedError
);
```

The *RasDialFunc1* function is just like the *RasDialFunc* function discussed earlier except that it features two additional parameters: *hrasconn* and *dwExtendedError*. The *hrasconn* parameter is the handle to the connection that *RasDial* returned. The *dwExtendedError* parameter allows you to retrieve extended error information when the following types of errors occur in the *dwError* parameter during the connection.

- *ERROR_SERVER_NOT_RESPONDING dwExtendedError* receives a NetBIOS-specific error code that occurred.
- *ERROR_NETBIOS_ERROR dwExtendedError* receives a NetBIOS-specific error code that occurred.
- *ERROR_AUTH_INTERNAL dwExtendedError* receives an internal diagnostic error code. These error codes are not documented.
- *ERROR_CANNOT_GET_LANA dwExtendedError* receives a routing RAS-specific error code.

The *RasDialFunc2* function is just like *RasDialFunc1* except that it features two additional parameters: *dwCallbackId* and *dwSubEntry*. The *dwCallbackId* parameter contains an application-defined value that was originally set in the *dwCallbackId* field of a *RASDIALPARAMS* structure passed to the *RasDial* call we described earlier. The *dwSubEntry* parameter receives the subentry phonebook index that caused the callback to the *RasDialFunc2* function.

Status Notification

RAS features a stand-alone function named *RasConnectionNotification* that allows your application to determine when an asynchronous RAS connection has been created or terminated. *RasConnectionNotification* is defined as

```
DWORD RasConnectionNotification(
    HRASCONN hrasconn,
    HANDLE hEvent,
    DWORD dwFlags
);
```

The *hrasconn* parameter is a connection handle returned from *RasDial*. The *hEvent* parameter is an event handle that your application creates using the *CreateEvent* function. The *dwFlags* parameter can be set to a combination of the following connection activity flags:

- *RASCN_Connection* Notifies you that a RAS connection has been created. If the *hrasconn* parameter is set to *INVALID_HANDLE_VALUE*, the event is signaled when any RAS connection occurs.
- *RASCN_Disconnection* Notifies you that a RAS connection has been terminated. If the *hrasconn* parameter is set to *INVALID_HANDLE_VALUE*, the event is signaled when any connection ends.
- *RASCN_BandwidthAdded* On a multilink connection, the event is signaled when a subentry connects.
- *RASCN_BandwidthRemoved* On a multilink connection, the event is signaled when a subentry disconnects.

Note that these flags function in the same way as the connection activity flags described in Table 16-3. If any of these activities occur during your connection, your event will become signaled. Your application should use Win32 wait functions, such as *WaitForSingleObject*, to determine when the object becomes signaled.

Closing a Connection

Closing a connection established by *RasDial* is simple. All you have to do is call *RasHangUp*, which is defined as

```
DWORD RasHangUp(
    HRASCONN hrasconn
);
```

The *hrasconn* parameter is a handle returned from *RasDial*. Although this function is easy to use, you have to consider how connections are managed internally in RAS. A connection uses a modem port, and it takes time for the port to reset internally when a connection shuts down. Therefore, you should wait until the port connection closes completely. To do this, you can call *RasGetConnectionStatus* to determine when your connection is reset. *RasGetConnectionStatus* is defined as

```
DWORD RasGetConnectStatus(
    HRASCONN hrasconn,
    LPRASCONNSTATUS lprasconnstatus
);
```

The *hrasconn* parameter is a handle returned from *RasDial*. The *lprasconnstatus* parameter is a *RASCONNSTATUS* structure that receives the current connection status. A *RASCONNSTATUS* structure is defined as

```
typedef struct _RASCONNSTATUS
{
    DWORD dwSize;
    RASCONNSTATE rasconnstate;
    DWORD dwError;
    TCHAR szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
} RASCONNSTATUS;
```

These fields are defined as follows:

- *dwSize* Should be set to the size (in bytes) of *RASCONNSTATUS*
- *rasconnstate* Receives one of the connection activities defined in Table 16-3
- *dwError* Receives a specific RAS error code if *RasGetConnectStatus* does not return 0
- *szDeviceType* Receives a string representing the type of device used on the connection
- *szDeviceName* Receives the name of the current device

We recommend that you check the state of your connection until you receive the *RASCS_ Disconnected* activity status. Obviously, you might have to call *RasGetConnectionStatus* several times until the connection is reset. Once the connection is reset, you can exit your application or make another connection.

Phonebook

RAS can set up communication to remote servers by using phonebook entries to store and manage properties for establishing a connection. A phonebook is nothing more than a collection of *RASENTRY* structures that contain phone numbers, data rates, user authentication information, and other connection information. On Windows 95, Windows 98, and Windows CE, the phonebook is stored in the system Registry. On Windows 2000 and Windows NT, the phonebook is stored in files that typically have the file extension .pbk. A *RASENTRY* structure is defined as

```
typedef struct tagRASENTRY
{
    DWORD     dwSize;
    DWORD     dwfOptions;
    DWORD     dwCountryID;
    DWORD     dwCountryCode;
    TCHAR     szAreaCode[RAS_MaxAreaCode + 1];
    TCHAR     szLocalPhoneNumber[RAS_MaxPhoneNumber + 1];
    DWORD     dwAlternateOffset;
    RASIPADDR ipaddr;
    RASIPADDR ipaddrDns;
    RASIPADDR ipaddrDnsAlt;
    RASIPADDR ipaddrWins;
    RASIPADDR ipaddrWinsAlt;
    DWORD     dwFrameSize;
    DWORD     dwfNetProtocols;
    DWORD     dwFramingProtocol;
    TCHAR     szScript[MAX_PATH];
    TCHAR     szAutodialDll[MAX_PATH];
    TCHAR     szAutodialFunc[MAX_PATH];
    TCHAR     szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR     szDeviceName[RAS_MaxDeviceName + 1];
    TCHAR     szX25PadType[RAS_MaxPadType + 1];
    TCHAR     szX25Address[RAS_MaxX25Address + 1];
    TCHAR     szX25Facilities[RAS_MaxFacilities + 1];
    TCHAR     szX25UserData[RAS_MaxUserData + 1];
    DWORD     dwChannels;
    DWORD     dwReserved1;
    DWORD     dwReserved2;
#if (WINVER >= 0x401)
    DWORD     dwSubEntries;
    DWORD     dwDialMode;
    DWORD     dwDialExtraPercent;
    DWORD     dwDialExtraSampleSeconds;
    DWORD     dwHangUpExtraPercent;
    DWORD     dwHangUpExtraSampleSeconds;
    DWORD     dwIdleDisconnectSeconds;

#endif
#if (WINVER >= 0x500)
    DWORD     dwType;
    DWORD     dwEncryptionType;
    DWORD     dwCustomAuthKey;
```

```
    GUID      guidId;
    TCHAR     szCustomDialDll[MAX_PATH];
    DWORD     dwVpnStrategy;
#endif
} RASENTRY;
```

As you can see, many fields make up this structure. The fields are defined as follows:

- *dwSize* Identifies the size (in bytes) of a *RASENTRY* structure.
- *dwfOptions* Can be set to one or more flags described in Table 16-4.

Table 16-4. RASENTRY *structure option flags*

Option Flag	*Description*
RASEO_Custom	On Windows 2000, custom encryption is used.
RASEO_DisableLcpExtensions	RAS disables the PPP LCP extensions defined in RFC 1570.
RASEO_IpHeaderCompression	RAS will negotiate IP header compression over a PPP connection.
RASEO_ModemLights	On Windows 2000, the task bar will display a status monitor.
RASEO_NetworkLogon	On Windows 95 and Windows 98, RAS will attempt to log the user on to a domain only after the RAS connection is authenticated.
RASEO_PreviewDomain	On Windows 2000, RAS will display the user's domain before dialing.
RASEO_PreviewPhoneNumber	On Windows 2000, RAS will display the phone number to be dialed.
RASEO_PromoteAlternates	RAS can make an alternate number the primary number if the alternate number connects successfully.
RASEO_RemoteDefaultGateway	The default route for IP packets is via the dial-up adapter rather than via other network adapters when the RAS connection is active.
RASEO_RequireCHAP	On Windows 2000, the Challenge Handshake Authentication Protocol (CHAP) will be used for authentication.
RASEO_RequireDataEncryption	Data encryption must be negotiated successfully; otherwise, the connection should be dropped. The *RASEO_RequireEncryptedPw* flag also must be set.
RASEO_RequireEAP	On Windows 2000, EAP will be used for authentication.
RASEO_RequireEncryptedPw	This flag prevents PPP from using the Password Authentication Protocol (PAP) to authenticate the client. The CHAP and Shiva's Password Authentication Protocol (SPAP) are used instead.
RASEO_RequireMsCHAP	On Windows 2000, Microsoft CHAP will be used for authentication.
RASEO_RequireMsCHAP2	On Windows 2000, Microsoft CHAP version 2 will be used for authentication.
RASEO_RequireMsEncryptedPw	This flag overrides *RASEO_RequireEncryptedPw* and allows RAS to use Microsoft secure password schemes such as Microsoft CHAP.
RASEO_RequirePAP	On Windows 2000, PAP will be used for authentication.
RASEO_RequireSPAP	On Windows 2000, SPAP will be used for authentication.
RASEO_RequireW95MSCHAP	On Windows 2000, an older version of Microsoft CHAP (designed for Windows 95 RAS servers) is used for authentication.

RASEO_ReviewUserPW	On Windows 2000, RAS will display the user's name and password before dialing.
RASEO_SecureLocalFiles	On Windows 2000 and Windows NT, RAS checks for existing remote file system and remote printer bindings before making a connection with this entry.
RASEO_SharedPhoneNumbers	On Windows 2000, phone numbers are shared.
RASEO_ShowDialingProgress	On Windows 2000, RAS will display dialing progress.
RASEO_SpecificIpAddr	This flag tells RAS to use the IP address specified in the *ipaddr* field.
RASEO_SpecificNameServers	This flag tells RAS to use the IP information specified in the *ipaddrDns*, *ipaddrDnsAlt*, *ipaddrWins*, and *ipaddrWinsAlt* fields.
RASEO_SwCompression	This flag lets RAS negotiate software compression of data sent over a connection.
RASEO_TerminalAfterDial	RAS displays a terminal window for user input after dialing the connection.
RASEO_TerminalBeforeDial	RAS displays a terminal window for user input before dialing the connection.
RASEO_UseCountryAndAreaCodes	This flag tells RAS to use the *dwCountryID*, *dwCountryCode*, and *szAreaCode* fields to construct a phone number with the *szLocalPhoneNumber* field.
RASEO_UseLogonCredentials	This flag tells RAS to use the user name, password, and domain of the user who is currently logged on when dialing. This occurs only if the *RASEO_RequireMsEncryptedPw* flag is set.

- *dwCountryID* Specifies a TAPI country identifier if the *RASEO_UseCountryAndAreaCodes* option flag is set. You can retrieve country identifier information by calling the *RasGetCountryInfo* function (which we'll describe later).
- *dwCountryCode* Specifies a country code that is associated with the *dwCountryID* field if the *RASEO_UseCountryAndAreaCodes* option flag is set. If this field is 0, the country code associated with *dwCountryID* in Windows is used.
- *szAreaCode* Specifies an area code if the *RASEO_UseCountryAndAreaCodes* flag is set.
- *szLocalPhoneNumber* Specifies a phone number for dialing. If the *RASEO_UseCountryAndAreaCodes* flag is set, RAS will combine the values of the *dwCountryID*, *dwCountryCode*, and *szAreaCode* fields with your phone number.
- *dwAlternateOffset* Specifies an offset in bytes, from the beginning of this structure, where alternate phone numbers are stored for a RAS phonebook entry. Alternate numbers are stored as a consecutive set of null-terminated strings. The last string in the set is terminated by two consecutive null characters.
- *ipaddr* Specifies an IP address to use for this connection if the *RASEO_SpecificIpAddr* flag is set.
- *ipaddrDns* Specifies a DNS server IP address for this connection if the *RASEO_SpecificNameServers* flag is set.
- *ipaddrDnsAlt* Specifies a secondary DNS server IP address for this connection if the *RASEO_SpecificNameServers* flag is set.
- *ipaddrWins* Specifies a WINS server IP address for this connection if the *RASEO_SpecificNameServers* flag is set.
- *ipaddrWinsAlt* Specifies a secondary WINS server IP address for this connection if the *RASEO_SpecificNameServers* flag is set.
- *dwFrameSize* Changes the size of the framing protocol to 1006 or 1500 bytes if the *RASFP_Slip* flag is set in the *dwFramingProtocol* field.

- *dwfNetProtocols* Specifies the flags identifying which network protocols will be used over the framing protocol. The protocol flags are *RASNP_NetBEUI* for NetBEUI, *RASNP_Ipx* for IPX, and *RASNP_Ip* for IP.

- *dwFramingProtocol* Specifies the flags identifying which framing protocol will be used over the RAS connection. The possible flags are *RASFP_Ppp* for PPP, *RASFP_Slip* for SLIP, and *RASFP_Ras* for asynchronous NetBEUI.

- *szScript* Specifies a full path to a dial-up script that gets executed when starting the connection.

- *szAutodialDll* Specifies a customizable DLL that can be used to set automatic dialing (autodial) features of RAS. This book does not cover the details of managing RAS autodial features.

- *szAutodialFunc* Specifies the function name exported from the customized DLL in the *szAutodialDll* field.

- *szDeviceType* Specifies a device type that is used for a connection. The value should be identified as a string. Table 16-5 lists the possible values.

Table 16-5. *RAS device types*

Device Name String	*Description*
"RASDT_Modem"	Modem on a COM port
"RASDT_Isdn"	ISDN adapter
"RASDT_X25"	X.25 card
"RASDT_Vpn"	Virtual private networking (VPN) connection
"RASDT_Pad"	Packet assembler/disassembler
"RASDT_Generic"	Generic type
"RASDT_Serial"	Serial port
"RASDT_FrameRelay"	Frame relay device
"RASDT_Atm"	ATM device
"RASDT_Sonet"	Synchronous Optical Network (Sonet) device
"RASDT_SW56"	Switched 56-Kbps access
"RASDT_Irda"	IrDA device
"RASDT_Parallel"	Parallel port

- *szDeviceName* Identifies a TAPI device to use for a connection. You can retrieve TAPI devices using the *RasEnumDevices* function, which we will describe later.

- *szX25PadType* Specifies an X.25 PAD type.

- *szX25Address* Specifies an X.25 address.

- *szX25Facilities* Specifies facilities to request from the X.25 host.

- *szX25UserData* Specifies extra information for an X.25 connection.

- *dwChannels* Is not used.

- *dwReserved1* Is not used and must be set to 0.

- *dwReserved2* Is not used and must be set to 0.

- *dwSubEntries* Identifies how many multilink subentries are associated with this phonebook entry. You should set this field to 0 and let the *RasSetSubEntryProperties* function manage multilink subentries for this

field. (More on this function later in this chapter.)

- *dwDialMode* Specifies in Windows 2000 how RAS should dial multilink subentries when a connection is made using the defines *RASEDM_DialAll* and *RASEDM_DialAsNeeded*. If this field is set to *RASEDM_DialAll*, all multilink subentries are dialed. If *RASEDM_DialAsNeeded* is set, RAS uses the *dwDialExtraPercent*, *dwDialExtraSampleSeconds*, *dwHangUpExtraPercent*, and *dwHangUpExtraSampleSeconds* fields to determine when to dial and disconnect additional multilink subentries.

- *dwDialExtraPercent* Specifies in Windows 2000 a percentage of the current total bandwidth of a connection. RAS will dial an additional subentry when the total bandwidth used exceeds this percentage for at least *dwDialExtraSampleSeconds*.

- *dwDialExtraSampleSeconds* Specifies in Windows 2000 the number of seconds by which current bandwidth usage must exceed the bandwidth percentage specified in *dwDialExtraPercent* before RAS dials an additional subentry.

- *dwHangUpExtraPercent* Specifies in Windows 2000 the percentage of total bandwidth available from the connected subentries. RAS will terminate subentry connections when the total bandwidth usage falls below this percentage for *dwHangUpExtraSampleSeconds*.

- *dwHangUpExtraSampleSeconds* Specifies in Windows 2000 the number of seconds by which current bandwidth usage must fall below the bandwidth percentage specified in *dwHangUpExtraPercent* before RAS disconnects a subentry.

- *dwIdleDisconnectSeconds* Specifies how many seconds of idle time are allowed before RAS will terminate the connection. You can also set this field to either *RASIDS_Disabled* to prevent the connection from terminating or *RASIDS_UseGlobalValue* to use the system's default value.

- *dwType* Specifies in Windows 2000 the type of phonebook entry. Table 16-6 lists the possible values.

Table 16-6. RASENTRY *phonebook types*

Type	*Description*
RASET_Direct	Direct serial or parallel connection
RASET_Internet	Internet connection services (ICS)
RASET_Phone	Phone line
RASET_Vpn	Virtual private network

- *dwEncryptionType* Specifies in Windows 2000 the type of encryption used on the data passed through the connection. Table 16-7 lists the possible values.

Table 16-7. *Data encryption values used on a RAS connection*

Value	*Description*
ET_40Bit	40-bit data encryption
ET_128Bit	128-bit data encryption

- *dwCustomAuthKey* Specifies on Windows 2000 an authentication key provided to the EAP vendor.

- *guidId* On Windows 2000, identifies a GUID associated with the phonebook entry.

- *szCustomDialDll* Specifies on Windows 2000 a path to a DLL containing custom RAS dialer functions. If this field is *NULL*, RAS will use the default system dialer. This book does not describe the details of developing a custom dialer for RAS.

- *dwVpnStrategy* Specifies on Windows 2000 the VPN dialing strategy to use on VPN connections. Table 16-8 lists the possible values.

Table 16-8. *RAS VPN dialing strategies*

Value	*Description*
VS_Default	RAS dials Point-to-Point Tunneling Protocol (PPTP) first. If PPTP fails, Layer 2 Tunneling Protocol (L2TP) is attempted.
VS_L2tpFirst	RAS dials L2TP first.
VS_L2tpOnly	RAS dials only L2TP.
VS_PptpFirst	RAS dials PPTP first.
VS_PptpOnly	RAS dials only PPTP.

When you call any RAS API that takes a phonebook file as a parameter (*lpszPhonebook*), you can identify the path to a phonebook file. As we mentioned earlier, this parameter must be *NULL* on Windows 95, Windows 98, and Windows CE because phonebook entries are stored in the system Registry. On Windows 2000 and Windows NT, this can be a path to a phonebook file. Typically, this phonebook file will have the extension .pbk. Also, the system default phonebook on Windows 2000 and Windows NT is located under %SystemRoot%\System32\Ras\RasPhone.pbk. If you specify *NULL* as the phonebook, you will use the system default phonebook file.

Three support functions can help you create and manage phonebook entries: *RasValidateEntryName*, *RasEnumDevices*, and *RasGetCountryInfo*. The *RasValidateEntryName* function, shown below, determines whether a name is properly formatted and whether it already exists in a phonebook.

```
DWORD RasValidateEntryName(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry
);
```

The *lpszPhonebook* parameter is a pointer to a name of a phonebook file. The *lpszEntry* parameter is a string representing the phonebook entry name you are verifying. This function returns *ERROR_SUCCESS* if the name is not already in the phonebook and is properly formatted. Otherwise, the function fails—with *ERROR_INVALID_NAME* if the name is not correctly formatted; with *ERROR_ALREADY_EXISTS* if the name is in the phonebook.

The *RasEnumDevices* function receives the name and type of all RAS-capable devices available on your computer:

```
DWORD RasEnumDevices(
    LPRASDEVINFO lpRasDevInfo,
    LPDWORD lpcb,
    LPDWORD lpcDevices
);
```

The *lpRasDevInfo* parameter is a pointer to an application buffer that you must provide to receive an array of *RASDEVINFO* structures. The *RASDEVINFO* structure is defined as

```
typedef struct tagRASDEVINFO {
    DWORD dwSize;
    TCHAR szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
} RASDEVINFOW;
```

The fields of this parameter are defined as follows:

- *dwSize* Must be set to the size (in bytes) of a *RASDEVINFO* structure before you call *RasEnumDevices*
- *szDeviceType* Receives a string describing the device type, such as "RASDT_Modem"
- *szDeviceName* Receives the formal name of a TAPI device

You must be sure you provide a buffer that is large enough to hold several structures; otherwise, *RasEnumDevices* will fail with the error *ERROR_BUFFER_TOO_SMALL*. The next parameter, *lpcb*, is a pointer to a variable that receives the number of bytes needed to enumerate the devices. You must set this parameter to the size (in bytes) of your *lpRasDevInfo* buffer. The final parameter, *lpcDevices*, is a pointer to a variable that receives the number of *RASDEVINFO* structures written to *lpRasDevInfo*.

The *RasGetCountryInfo* function allows you to retrieve country-specific TAPI dialing information from Windows:

```
DWORD RasGetCountryInfo(
    LPRASCTRYINFO lpRasCtryInfo,
    LPDWORD lpdwSize
);
```

The *lpRasCtryInfo* parameter is a buffer that receives a dialing prefix and other information associated with the country you specify. This buffer must be a *RASCTRYINFO* structure that is followed by additional bytes that receive a country description string. We recommend allocating at least a 256-byte buffer to hold the *RASCTRYINFO* structure and the description string. The *RASCTRYINFO* structure is defined as

```
typedef struct RASCTRYINFO
{
    DWORD dwSize;
    DWORD dwCountryID;
    DWORD dwNextCountryID;
    DWORD dwCountryCode;
    DWORD dwCountryNameOffset;
} RASCTRYINFO;
```

The fields are described as follows:

- *dwSize* Must be set to the size (in bytes) of a *RASCTRYINFO* structure.
- *dwCountryID* Allows you to specify a TAPI country identifier (which applies to the *dwCountryID* field of a *RASENTRY* structure) in the Windows list of countries. If you set this field to 1, you will receive the first entry in the list.

- *dwNextCountryID* Receives the next TAPI country identifier in the list. If this field is set to 0, you are at the end of the list.
- *dwCountryCode* Receives the dialing prefix code associated with the country specified in the *dwCountryID* parameter.
- *dwCountryNameOffset* Specifies the number of bytes from the start of this structure to the start of the null-terminated string describing the country that follows the *RASCTRYINFO* structure.

The other parameter of *RasGetCountryInfo*, *lpdwSize*, is a pointer to a variable that receives the number of bytes that *RasGetCountryInfo* placed in the *lpRasCtryInfo* buffer. You must set this parameter to the size of your application buffer before calling this function.

Adding Phonebook Entries

RAS provides four functions that allow you to programmatically manage phonebook *RASENTRY* structures: *RasSetEntryProperties*, *RasGetEntryProperties*, *RasRenameEntry*, and *RasDeleteEntry*. You can use the *RasSetEntryProperties* function, which is defined below, to create a new entry or modify an existing entry.

```
DWORD RasSetEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASENTRY lpRasEntry,
    DWORD dwEntryInfoSize,
    LPBYTE lpbDeviceInfo,
    DWORD dwDeviceInfoSize
);
```

The *lpszPhonebook* parameter is a pointer to the name of a phonebook file. The *lpszEntry* parameter is a pointer to a string used to identify an existing entry or a new one. If a *RASENTRY* structure exists with this name, the properties are modified; otherwise, a new entry is created in the phonebook. The *lpRasEntry* parameter is a pointer to a *RASENTRY* structure. You can place a list of null-terminated strings after the *RASENTRY* structure defining alternate phone numbers. The last string is terminated by two consecutive null characters. The *dwEntryInfoSize* parameter is the size (in bytes) of the structure in the *lpRasEntry* parameter. The *lpbDeviceInfo* parameter is a pointer to a buffer that contains TAPI device configuration information. On Windows 2000 and Windows NT, this parameter is not used and should be set to *NULL*. The final parameter, *dwDeviceInfoSize*, represents the size (in bytes) of the *lpbDeviceInfo* buffer.

The *RasGetEntryProperties* function, defined below, can be used to retrieve the properties of an existing phonebook entry or the default values for a new phonebook entry.

```
DWORD RasGetEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASENTRY lpRasEntry,
    LPDWORD lpdwEntryInfoSize,
    LPBYTE lpbDeviceInfo,
    LPDWORD lpdwDeviceInfoSize
);
```

The *lpszPhonebook* parameter is a pointer to the name of a phonebook file. The *lpszEntry* parameter is a pointer to a string identifying an existing phonebook entry. If you set this parameter to *NULL*, the *lpRasEntry* and *lpbDeviceInfo* parameters will receive default values of a phonebook entry. Retrieving the default values is quite useful: when you need to create a new RAS phonebook entry, you can populate the *lpRasEntry* and *lpbDeviceInfo*

fields with correct information about your system before you call the *RasSetEntryProperties* function.

The *lpRasEntry* parameter is a pointer to a buffer that your application supplies to receive a *RASENTRY* structure. As we described in our discussion of the *RasSetEntryProperties* function, this structure can be followed by an array of null-terminated strings identifying alternate phone numbers for the requested phonebook entry. Therefore, the size of your receiving buffer should be larger than a *RASENTRY* structure. If you pass a *NULL* pointer, the *lpdwEntryInfoSize* parameter will receive the total number of bytes needed to store all the elements of a *RASENTRY* structure plus any alternate phone numbers. The *lpdwEntryInfoSize* parameter is a pointer to a *DWORD* containing the number of bytes that are in the receiving buffer supplied by your application to the *lpRasEntry* parameter. When this function completes, it will update *lpdwEntryInfoSize* to the number of bytes actually received in *lpRasEntry*. We highly recommend calling this function with *lpRasEntry* set to *NULL* and *lpdwEntryInfoSize* set to 0 to obtain buffer sizing information. Once you have the appropriate size, you can call this function again and retrieve all the information without error.

The *lpbDeviceInfo* parameter is a pointer to an application-supplied buffer that receives TAPI device-specific information for this phonebook entry. If this parameter is set to *NULL*, the *lpdwDeviceInfoSize* parameter will receive the number of bytes needed to retrieve this information. If you are using Windows 2000 and Windows NT, *lpbDeviceInfo* should be set to *NULL*. The final parameter, *lpdwDeviceInfoSize*, is a pointer to a *DWORD* that should be set to the number of bytes contained in the buffer supplied to *lpbDeviceInfo*. When *RasGetEntryProperties* returns, *lpdwDeviceInfoSize* will return the number of bytes that are returned in the *lpbDeviceInfo* buffer.

Figure 16-3 demonstrates how an application should use *RasGetEntryProperties* and *RasSetEntryProperties* to create a new phonebook entry.

Figure 16-3. *Creating a new RAS phonebook entry using default properties*

```
#include <windows.h>
#include <ras.h>
#include <raserror.h>
#include <stdio.h>

void main(void)
{
    DWORD EntryInfoSize = 0;
    DWORD DeviceInfoSize = 0;
    DWORD Ret;
    LPRASENTRY lpRasEntry;
    LPBYTE lpDeviceInfo;

    // Get buffer sizing information for a default phonebook entry

    if ((Ret = RasGetEntryProperties(NULL, "", NULL,
        &EntryInfoSize, NULL, &DeviceInfoSize)) != 0)
    {
        if (Ret != ERROR_BUFFER_TOO_SMALL)
        {
            printf("RasGetEntryProperties sizing failed "
                "with error %d\n", Ret);
            return;
        }
    }

    lpRasEntry = (LPRASENTRY) GlobalAlloc(GPTR, EntryInfoSize);

    if (DeviceInfoSize == 0)
        lpDeviceInfo = NULL;
```

```
    else
        lpDeviceInfo = (LPBYTE) GlobalAlloc(GPTR, DeviceInfoSize);

    // Get default phonebook entry
    lpRasEntry->dwSize = sizeof(RASENTRY);

    if ((Ret = RasGetEntryProperties(NULL, "", lpRasEntry,
        &EntryInfoSize, lpDeviceInfo, &DeviceInfoSize)) != 0)
    {
        printf("RasGetEntryProperties failed with error %d\n",
            Ret);
        return;
    }

    // Validate new phonebook name "Testentry"

    if ((Ret = RasValidateEntryName(NULL, "Testentry")) !=
        ERROR_SUCCESS)
    {
        printf("RasValidateEntryName failed with error %d\n",
            Ret);
        return;
    }

    // Install a new phonebook entry, "Testentry", using
    // default properties

    if ((Ret = RasSetEntryProperties(NULL, "Testentry",
        lpRasEntry, EntryInfoSize, lpDeviceInfo,
        DeviceInfoSize)) != 0)
    {
        printf("RasSetEntryProperties failed with error %d\n",
            Ret);
        return;
    }
}
```

Renaming Phonebook Entries

Now that you have an understanding of what's involved in creating and modifying a phonebook entry, let's look at the *RasRenameEntry* function. *RasRenameEntry*, which is defined below, simply allows you to rename a phonebook entry:

```
DWORD RasRenameEntry(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszOldEntry,
    LPCTSTR lpszNewEntry
);
```

The *lpszPhonebook* parameter is a pointer to a name of a phonebook file. The *lpszOldEntry* parameter is a pointer to a string identifying an existing phonebook entry that you intend to rename. The *lpszNewEntry* parameter is a pointer to a string containing the new name for the phonebook entry. Your application should call *RasValidateEntryName* on the new name before calling *RasRenameEntry*. If *RasRenameEntry* succeeds, it will return 0. If it fails, it will return the following types of errors:

- *ERROR_INVALID_NAME* Indicates that the *lpszNewEntry* name is invalid
- *ERROR_ALREADY_EXISTS* Indicates that the *lpszNewEntry* name already exists in the phonebook
- *ERROR_CANNOT_FIND_PHONEBOOK_ENTRY* Indicates that the *lpszOldEntry* name cannot be found in the phonebook

Deleting Phonebook Entries

Deleting phonebook entries is easy. To do so, you simply call the *RasDeleteEntry* function, which is defined as

```
DWORD RasDeleteEntry(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry
);
```

The *lpszPhonebook* parameter is a pointer to a name of a phonebook file. The *lpszEntry* parameter is a string representing an existing phonebook entry. If this function succeeds, it returns *ERROR_SUCCESS*; otherwise, it returns *ERROR_INVALID_NAME.*

Enumerating Phonebook Entries

RAS provides a convenient function named *RasEnumEntries*, which retrieves the phonebook entries available in a phonebook file and is defined as shown below.

```
DWORD RasEnumEntries (
    LPCTSTR reserved,
    LPCTSTR lpszPhonebook,
    LPRASENTRYNAME lprasentryname,
    LPDWORD lpcb,
    LPDWORD lpcEntries
);
```

The *reserved* parameter isn't used and must be set to *NULL*. The *lpszPhonebook* parameter is a pointer to the name of a phonebook file. The *lprasentryname* parameter is a pointer to an application buffer that you must provide to receive an array of *RASENTRYNAME* structures. The *RASENTRYNAME* structure is defined as

```
typedef struct _RASENTRYNAME
{
    DWORD dwSize;
    TCHAR szEntryName[RAS_MaxEntryName + 1];
#if (WINVER >= 0x500)
    DWORD dwFlags;
    CHAR  szPhonebookPath[MAX_PATH + 1];
#endif
} RASENTRYNAME;
```

The fields are defined as follows:

- *dwSize* Must be set to the size (in bytes) of a *RASENTRYNAME* structure before you call *RasEnumEntries*.
- *szEntryName* Receives the name of a phonebook entry.
- *dwFlags* On Windows 2000, this flag indicates whether the phonebook entry is in the system default phonebook (described earlier) using flag *REN_AllUsers* or is in a user's profile phonebook using flag *REN_User*.
- *szPhonebookPath* On Windows 2000, specifies the full path to the phonebook file.

You must be sure you provide a buffer large enough to hold several structures; otherwise, *RasEnumEntries* will fail with the error *ERROR_BUFFER_TOO_SMALL*. The next parameter, *lpcb*, is a pointer to a variable that receives the number of bytes needed to enumerate the entries. You must set this parameter to the size (in bytes) of your *lprasentryname* buffer. The final parameter, *lpcEntries*, is a pointer to a variable that receives the number of *RASENTRYNAME* structures written to the *lprasentryname* buffer.

Managing User Credentials

When a RAS client makes a connection using a phonebook entry through *RasDial*, it saves the security credentials of the user and associates them with the phonebook entry. The functions *RasGetCredentials*, *RasSetCredentials*, *RasGetEntryDialParams*, and *RasSetEntryDialParams* allow you to manage user security credentials associated with a phonebook entry. The *RasGetCredentials* and *RasSetCredentials* functions were introduced in Windows NT 4. (They are also available on Windows 2000.) These two functions supersede *RasGetEntryDialParams* and *RasSetEntryDialParams*. Since *RasGetCredentials* and *RasSetCredentials* are not available on Windows 95, Windows 98, and Windows CE, you can use *RasGetEntryDialParams* and *RasSetEntryDialParams* for this purpose on all platforms.

The *RasGetCredentials* function, defined below, retrieves user credentials associated with a phonebook entry:

```
DWORD RasGetCredentials(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASCREDENTIALS lpCredentials
);
```

The *lpszPhonebook* parameter is a pointer to the name of a phonebook file. The *lpszEntry* parameter is a string representing an existing phonebook entry. The *lpCredentials* parameter, defined below, is a pointer to a *RASCREDENTIALS* structure that can receive the user name, password, and domain associated with the phonebook entry:

```
typedef struct {
    DWORD dwSize;
    DWORD dwMask;
    TCHAR szUserName[UNLEN + 1];
    TCHAR szPassword[PWLEN + 1];
    TCHAR szDomain[DNLEN + 1];
} RASCREDENTIALS, *LPRASCREDENTIALS;
```

The fields of this structure are defined as follows:

- *dwSize* Specifies the size (in bytes) of a *RASCREDENTIALS* structure. You should always set this field to the structure size.
- *dwMask* Is a bitmask field that identifies which of the next three fields in the structure are valid by using predefined flags. The flag *RASCM_UserName* applies to *szUserName*, *RASCM_Password* applies to *szPassword*, and *RASCM_Domain* applies to *szDomain*.
- *szUserName* Is a null-terminated string containing a user's logon name.
- *szPassword* Is a null-terminated string containing a user's password.
- *szDomain* Is a null-terminated string containing a user's logon domain.

If *RasGetCredentials* succeeds, it returns 0. Your application can determine which security credentials are set based on the flags set in the *dwMask* field of the *lpCredentials* structure.

The *RasSetCredentials* function is similar to *RasGetCredentials* except that it lets you change security credentials associated with a phonebook entry. The parameters are the same except that *RasSetCredentials* features an additional parameter: *fClearCredentials*. *RasSetCredentials* is defined as

```
DWORD RasSetCredentials(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASCREDENTIALS lpCredentials,
    BOOL fClearCredentials
);
```

The *fClearCredentials* parameter is a Boolean operator that, if set to *TRUE*, causes *RasSetCredentials* to change credentials identified in the *dwMask* field of the *lpCredentials* structure to an empty string ("") value. For example, if *dwMask* contains the *RASCM_Password* flag, the password stored is replaced with an empty string. If the *RasSetCredentials* function succeeds, it returns 0.

You can also use *RasGetEntryDialParams* and *RasSetEntryDialParams* to manage user security credentials associated with phonebook entries. *RasGetEntryDialParams* is defined as

```
DWORD RasGetEntryDialParams(
    LPCTSTR lpszPhonebook,
    LPRASDIALPARAMS lprasdialparams,
    LPBOOL lpfPassword
);
```

The *lpszPhonebook* parameter is a pointer to the name of a phonebook file. The *lprasdialparams* parameter is a pointer to a *RASDIALPARAMS* structure. The *lpfPassword* parameter is a Boolean flag that returns *TRUE* if the user's password was retrieved in the *lprasdialparams* structure.

The *RasSetEntryDialParams* function changes the connection information that was last set by the *RasDial* call on a particular phonebook entry. *RasSetEntryDialParams* is defined as

```
DWORD RasSetEntryDialParams(
    LPCTSTR lpszPhonebook,
    LPRASDIALPARAMS lprasdialparams,
    BOOL fRemovePassword
);
```

The *lpszPhonebook* and *lprasdialparams* parameters are the same as the first two parameters in *RasGetEntryDialParams*. The *fRemovePassword* parameter is a Boolean flag that, if set to *TRUE*, tells *RasSetEntryDialParams* to remove the password associated with the phonebook entry identified in the *lprasdialparams* structure.

Multilink Phonebook Subentries

On Windows 2000 and Windows NT, RAS allows you to manage multilink phonebook entries for enhanced communication capability. Multilink entries enable you to have more than one communication device associated with a RAS connection to increase the connection's total bandwidth. RAS allows you to manage multilink phonebook entries by using *RasGetSubEntryProperties* and *RasSetSubEntryProperties*. The *RasGetSubEntryProperties* function is defined as

```
DWORD RasGetSubEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    DWORD dwSubEntry,
    LPRASSUBENTRY lpRasSubEntry,
    LPDWORD lpdwcb,
    LPBYTE lpbDeviceConfig,
    LPDWORD lpcbDeviceConfig
);
```

The *lpszPhonebook* parameter is a pointer to the name of a phonebook file. The *lpszEntry* parameter is a phonebook entry. The *dwSubEntry* parameter specifies the index of a subentry contained within the phonebook entry. The *lpRasSubEntry* parameter is a pointer to a buffer that will receive a *RASSUBENTRY* structure followed by an optional list of alternate phone numbers. A *RASSUBENTRY* structure is defined as

```
typedef struct tagRASSUBENTRY
{
    DWORD dwSize;
    DWORD dwfFlags;
    TCHAR szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
    TCHAR szLocalPhoneNumber[RAS_MaxPhoneNumber + 1];
    DWORD dwAlternateOffset;
} RASSUBENTRY;
```

The fields are defined as follows:

- *dwSize* Must be set to the size (in bytes) of a *RASSUBENTRY* structure
- *dwFlags* Is not used
- *szDeviceType* Receives a string representing the type of device that is used on the connection
- *szDeviceName* Receives the actual name of the TAPI device
- *szLocalPhoneNumber* Identifies the phone number to use for this device
- *dwAlternateOffset* Specifies the number of bytes from the beginning of the structure to the list of consecutive null-terminated strings that follow the *RASSUBENTRY* structure

The *lpRasSubEntry* buffer must be large enough to contain a *RASSUBENTRY* structure along with alternate phone number strings; otherwise, the *RasGetSubEntryProperties* will fail with the error *ERROR_BUFFER_TOO_SMALL*. The *lpdwcb* parameter should be set to the number of bytes in your *lpRasSubEntry* buffer. On return, *lpdwcb* will receive the total number of bytes needed to contain a *RASSUBENTRY* structure along with alternate phone numbers. The *lpbDeviceConfig* and *lpcbDeviceConfig* parameters are not used and should be set to *NULL*.

You can create a new subentry or modify an existing subentry of a specified phonebook entry by calling the *RasSetSubEntryProperties* function, which is defined as

```
DWORD RasSetSubEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    DWORD dwSubEntry,
    LPRASSUBENTRY lpRasSubEntry,
    DWORD dwcbRasSubEntry,
    LPBYTE lpbDeviceConfig,
    DWORD dwcbDeviceConfig
);
```

The parameters are the same as those described by *RasGetSubEntryProperties* except that *lpRasSubEntry* specifies the subentry to add to the phonebook.

Connection Management

RAS has three useful functions that allow you to retrieve the properties of connections established on your system: *RasEnumConnections*, *RasGetSubEntryHandle*, and *RasGetProjectionInfo*. The *RasEnumConnections* function lists all active RAS connections and is defined as

```
DWORD RasEnumConnections(
    LPRASCONN lprasconn,
    LPDWORD lpcb,
    LPDWORD lpcConnections
);
```

The *lprasconn* parameter is an application buffer that will receive an array of *RASCONN* structures. A *RASCONN* structure is defined as

```
typedef struct _RASCONN
{
    DWORD dwSize;
    HRASCONN hrasconn;
    TCHAR szEntryName[RAS_MaxEntryName + 1];
#if (WINVER >= 0x400)
    CHAR szDeviceType[RAS_MaxDeviceType + 1];
    CHAR szDeviceName[RAS_MaxDeviceName + 1];
#endif
#if (WINVER >= 0x401)
    CHAR szPhonebook[MAX_PATH];
    DWORD dwSubEntry;
#endif
#if (WINVER >= 0x500)
    GUID guidEntry;
#endif
} RASCONN;
```

The fields are defined as follows:

- *dwSize* Indicates the size (in bytes) of a *RASCONN* structure.
- *hrasconn* Receives the connection handle that is created by *RasDial*.
- *szEntryName* Receives the phonebook entry that was used to establish the connection. If an empty string was used, the field will return a string with a period (.) followed by the phone number used.
- *szDeviceType* Receives a string describing the device type used in the connection.
- *szDeviceName* Receives a string with the name of the device that was used to make the connection.
- *szPhonebook* Receives the full path to the phonebook for the entry that made the connection.
- *dwSubEntry* Receives a subentry index of a multilink phonebook entry.

- *guidEntry* On Windows 2000, receives a GUID for the phonebook entry used to make the connection.

You need to pass to *RasEnumConnections* a large enough buffer to hold several *RASCONN* structures; otherwise, this function will fail with the error *ERROR_BUFFER_TOO_SMALL*. Furthermore, the first *RASCONN* structure in your buffer must have the *dwSize* field set to the byte size of a *RASCONN* structure. The next parameter, *lpcb*, is a pointer to a variable that you must set to the size (in bytes) of your *lprasconn* array. When this function returns, *lpcb* will contain the number of bytes required to enumerate all connections. If you don't supply a large enough buffer, you can always try again with the correct buffer size returned in *lpcb*. The *lpcConnections* parameter is a pointer to a variable that receives a count of the number of *RASCONN* structures written to *lprasconn*.

The *RasGetSubEntryHandle* function, defined as follows, allows you to retrieve a connection handle for a specified subentry of a multilink connection.

```
DWORD RasGetSubEntryHandle(
    HRASCONN hrasconn,
    DWORD dwSubEntry,
    LPHRASCONN lphrasconn
);
```

The *hrasconn* parameter is a RAS connection handle of a multilink connection. The *dwSubEntry* parameter is a subentry index of a device in the multilink connection. The *lphrasconn* parameter receives the connection handle for the subentry device.

With the RAS connection handles you receive from *RasEnumConnections* and *RasGetSubEntryHandle*, you can obtain network protocol-specific information that is used over an established RAS connection. This network protocol-specific information is known as *projection information.* A remote access server uses projection information to represent a remote client on the network. For example, when you make a RAS connection that uses the IP protocol over a framing protocol, IP configuration information (such as an assigned IP address) is established from the RAS service to your client. You can retrieve projection information for the protocols that travel over the PPP framing protocol by calling the *RasGetProjectionInfo* function, which is defined as

```
DWORD RasGetProjectionInfo(
    HRASCONN hrasconn,
    RASPROJECTION rasprojection,
    LPVOID lpprojection,
    LPDWORD lpcb
);
```

The *hrasconn* parameter is a RAS connection handle. The *rasprojection* parameter is a *RASPROJECTION* enumeration type that allows you to specify a protocol to receive connection information for. The *lpprojection* parameter receives a data structure that is associated with the enumeration type specified in *rasprojection*. The final parameter, *lpcb*, is a pointer to a variable that you must set to the size of your *lpprojection* structure. When this function completes, this variable will contain the size of the buffer needed to obtain the projection information.

The following *RASPROJECTION* enumeration types allow you to receive connection information:

- *RASP_Amb*
- *RASP_PppNbf*
- *RASP_PppIpx*
- *RASP_PppIp*

If you specify a *RASP_Amb* enumeration type, you will receive a *RASAMB* structure that is defined as

```
typedef struct _RASAMB
{
    DWORD dwSize;
    DWORD dwError;
    TCHAR szNetBiosError[NETBIOS_NAME_LEN + 1];
    BYTE bLana;
} RASAMB;
```

The fields are defined as follows:

- *dwSize* Should be set to the size (in bytes) of a *RASAMB* structure.
- *dwError* Receives an error code from the PPP negotiation process.
- *szNetBiosError* Receives a NetBIOS name if a name conflict occurs during the authentication process of PPP. If *dwError* returns *ERROR_NAME_EXISTS_ON_NET*, *szNetBiosError* will receive the name that caused the error.
- *bLana* Identifies the NetBIOS LAN adapter (LANA) number that was used to establish the remote access connection.

If you specify a *RASP_PppNbf* enumeration type to *RasGetProjectionInfo*, you will receive a *RASPPPNBF* structure that is defined as

```
typedef struct _RASPPPNBF
{
    DWORD dwSize;
    DWORD dwError;
    DWORD dwNetBiosError;
    TCHAR szNetBiosError[NETBIOS_NAME_LEN + 1];
    TCHAR szWorkstationName[NETBIOS_NAME_LEN + 1];
    BYTE bLana;
} RASPPPNBF;
```

The fields of *RASPPPNBF* are like the fields of the *RASAMB* structure except that *RASPPPNBF* contains two additional fields: *szWorkstationName* and *dwNetBiosError*. The *szWorkstationName* field receives the NetBIOS name that is used to identify your workstation on the network you are connecting to. The *dwNetBiosError* field receives the NetBIOS error that occurred.

If you specify a *RASP_PppIpx* enumeration type to *RasGetProjectionInfo*, you will receive a *RASPPPIPX* structure that is defined as

```
typedef struct _RASPPPIPX
{
    DWORD dwSize;
    DWORD dwError;
    TCHAR szIpxAddress[RAS_MaxIpxAddress + 1];
} RASPPPIPX;
```

The fields are defined as follows:

- *dwSize* Should be set to the size (in bytes) of a *RASPPPIPX* structure
- *dwError* Receives an error code from the PPP negotiation process
- *szIpxAddress* Receives a string representing the client's IPX address on the remote network

If you specify a *RASP_PppIp* enumeration type to *RasGetProjectionInfo*, you will receive a *RASPPPIP* structure that is defined as

```
typedef struct _RASPPPIP
{
    DWORD dwSize;
    DWORD dwError;
    TCHAR szIpAddress[RAS_MaxIpAddress + 1];
    TCHAR szServerIpAddress[RAS_MaxIpAddress + 1];
} RASPPPIP;
```

The fields are defined as follows:

- *dwSize* Should be set to the size (in bytes) of a *RASPPPIP* structure
- *dwError* Receives an error code from the PPP negotiation process
- *szIpAddress* Receives a string representing the client's IP address
- *szServerIpAddress* Receives a string representing the server's IP address

Figure 16-4 demonstrates how to retrieve the IP addresses assigned to an IP connection that is made over RAS.

Figure 16-4. *Using* RasGetProjectionInfo *on an IP connection*

```
lpProjection = (RASPPPIP *) GlobalAlloc(GPTR, cb);
lpProjection->dwSize = sizeof(RASPPPIP);
cb = sizeof(RASPPPIP);

Ret = RasGetProjectionInfo(hRasConn, RASP_PppIp,
    lpProjection, &cb);

if (Ret != ERROR_SUCCESS)
{
    printf("RasGetProjectionInfo failed with error %d", Ret);
    return;
}
else
{
    printf("\nRas Client IP address: %s\n",
        lpProjection->szIpAddress);
    printf("Ras Server IP address: %s\n",
        lpProjection->szServerIpAddress);
}
```

Conclusion

This chapter presented the basics of how to use RAS to extend your computer's networking capability. We described how to call the *RasDial* function to communicate with remote networks. We also discussed how to further utilize the capabilities of RAS by creating phonebook entries. This chapter concludes our discussion of RAS and Part III of this book.

Appendix A -- NetBIOS Command Reference

This appendix lists and describes the valid commands for the *ncb_command* field of the *NCB* structure that you must pass to the *Netbios* function. Each command description includes a table that indicates which fields of the *NCB* structure you must set for that command, and which fields the *Netbios* function sets prior to returning. Each table lists two columns. The first column indicates whether the given field of the *NCB* structure is an input or output parameter. The second column indicates whether the field must be set when making a NetBIOS call. If an X is present, a value must be provided. Otherwise, if the field is an input parameter and no X is present, providing a value is optional. Please refer to Chapter 1 for an in-depth discussion of the *Netbios* function.

NCBADDGRNAME

This command adds a group name to the local name table. This name cannot collide with a unique name, but anyone else can use it as a group name. Group names are most often used as recipients of datagrams. A name number is returned in the *ncb_num* field that is used in datagram operations.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num	OUT	
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name	IN	X
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBADDNAME

This command adds a unique name to the local name table. This name must be unique across the network, or an error is returned. A name number is returned in the *ncb_num* field that is used in datagram operations.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num	OUT	
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name	IN	X
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBASTAT

This command retrieves the status of a local or remote adapter. When you call this command, set *ncb_buffer* to point to a buffer that has an *ADAPTER_STATUS* structure followed by an array of *NAME_BUFFER* structures.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num		
ncb_buffer	IN/OUT	X
ncb_length	IN/OUT	X
ncb_callname	IN	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBCALL

This command connects (opens) a session to another process that you indicate in the *ncb_name* field.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn	OUT	
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname	IN	X
ncb_name	IN	X
ncb_rto	IN	
ncb_sto	IN	
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBCANCEL

This command cancels a previous outstanding command. The *ncb_buffer* field points to the *NCB* structure with the operation that you want canceled. Canceling an *NCBSEND* or *NCBCHAINSEND* command aborts the session; however, aborting their no-ack variants does not cancel their respective sessions. The following commands cannot be canceled: *NCBADDGRNAME*, *NCBADDNAME*, *NCBCANCEL*, *NCBDELNAME*, *NCBRESET*, *NCBDGSEND*, *NCBDGSENDBC*, and *NCBSSTAT*.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num		
ncb_buffer	IN	X
ncb_length		
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post		
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event		

NCBCHAINSEND

This command sends the contents of two buffers to the specified receiver. The maximum amount of data that can be sent is 128 KB (a maximum of 64 KB in each buffer). Use *ncb_buffer* and *ncb_length* to point to the first buffer and specify its length. Use bytes 0-1 of *ncb_callname* to specify the length of the second buffer, and use bytes 2-5 to point to it.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn	IN	X
ncb_num		
ncb_buffer	IN	X
ncb_length	IN	X
ncb_callname	IN	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBCHAINSENDNA

This command sends the contents of two buffers to the specified receiver and does not wait for any acknowledgment from the receiver. The maximum amount of data that can be sent is 128 KB (a maximum of 64 KB in each buffer). Specify the first buffer and its length in *ncb_buffer* and *ncb_length*, respectively. Use bytes 0-1 of *ncb_callname* to specify the length of the second buffer, and use bytes 2-5 to point to it.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn	IN	X
ncb_num		
ncb_buffer	IN	X
ncb_length	IN	X
ncb_callname	IN	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBDELNAME

This command deletes a name from the local name table. If the name to be deleted is associated with active sessions, the error *NRC_ACTSES* (0x0F) is returned. If any nonactive session commands are outstanding, they receive the error *NRC_NAMERR* (0x17).

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name	IN	X
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBDGRECV

This command receives a datagram directed to the local name associated with the *ncb_num* value. If *ncb_num* is 0xFF, this command receives datagrams directed to any local name. The local name can be either a group name or a unique name. If no receive datagram command is pending when a datagram is sent, the data is lost. If the supplied buffer is too small, the message "incomplete error," *NRC_INCOMP* (0x06), occurs and the data is truncated to fill the buffer.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num	IN	X
ncb_buffer	IN	X
ncb_length	IN/OUT	X
ncb_callname	OUT	
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBDGRECVBC

This command receives a broadcast datagram from any name issuing a command to send broadcast datagrams. The message "incomplete error," *NRC_INCOMP* (0x06), occurs if the supplied buffer is not large enough, and the data is truncated to fill the buffer.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num	IN	X
ncb_buffer	IN	X
ncb_length	IN/OUT	X
ncb_callname	OUT	
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBDGSEND

This command sends a datagram to a specified name. The name can be either a unique name or a group name. If an adapter has a pending receive datagram command for the same name, the adapter receives its own message. The maximum datagram size depends on the underlying protocol. To find the maximum datagram size, you can perform a local *NCBASTAT* command. The *ADAPTER_STATUS* structure that is returned will give the maximum datagram size for the underlying transport protocol.

Field	In/Out	Required
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num	IN	X
ncb_buffer	IN	X
ncb_length	IN	X
ncb_callname	IN	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBDGSENDBC

This command sends a broadcast datagram to every host on the LAN. Only those machines with an outstanding receive datagram command will get the message. Also, if the local adapter has a pending receive datagram command, it will receive its own message. Broadcast datagrams have the same size limitation mentioned in the *NCBDGSEND* entry.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num	IN	X
ncb_buffer	IN	X
ncb_length	IN	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBENUM

This command enumerates LANA numbers. When you issue this command, set *ncb_buffer* to a *LANA_ENUM* structure. On return, the *length* field of *LANA_ENUM* returns the number of LANA numbers on the local machine. The *lana* field of *LANA_ENUM* is filled with the LANA numbers.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num		
ncb_buffer	IN	X
ncb_length	IN	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post		
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event		

NCBFINDNAME

This command finds the location (machine name) of a name on the network. When this command is issued, *ncb_buffer* is filled with a *FIND_NAME_HEADER* structure, followed by one or more *FIND_NAME_BUFFER* structures. This command is Microsoft Windows NT-specific and is not supported on any other Win32 platforms.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num		
ncb_buffer	IN/OUT	X
ncb_length	IN	X
ncb_callname	IN	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBHANGUP

This command closes a specified connected session. All pending receive commands for the session are terminated and return with the "session closed" error, *NRC_SCLOSED* (0x0A). If either send or chain send commands are outstanding, the hang up command delays until the command completes. This delay occurs whether the commands are transferring data or waiting for the remote side to issue a receive command. Additionally, if multiple outstanding *NCBRECVANY* commands exist, only one of them will return an error code when the session is closed. For any other receive command, each outstanding receive returns an error.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn	IN	X
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBLANSTALERT

This is a Windows NT-only command that notifies the user of LAN failures that last for more than one minute. However, in testing, this command did nothing in response to several common LAN failures, such as disconnected network cables.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post		
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event		

NCBLISTEN

This command listens for a connection from another process, local or remote. If the first character of *ncb_callname* is an asterisk (*), a session is established with any network adapter that issues an *NCBCALL* to the local name. The name making the *NCBCALL* is returned in the *ncb_callname* field. If either a send or receive timeout is specified, these timeout values are applied to all send and receive calls made on the new session.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn	OUT	
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname	IN/OUT	X
ncb_name	IN	X
ncb_rto	IN	
ncb_sto	IN	
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBRECV

This command receives data from the specified session name. If more than one command capable of receiving data is pending, they are processed in the following order:

1. Receive (*NCBRECV*)
2. Receive-any for a specified name (*NCBRECVANY*)
3. Receive-any for any name (*NCBRECVANY*)

All commands with the same precedence are processed in first-in, first-out order. If the buffer passed is not large enough to hold the data, the error *NRC_INCOMP* (0x06) is returned. If this occurs, issue another receive command with a larger buffer unless the send command was issued with either a timeout that expired or a no-ack—in which case the data is lost. The *ncb_length* field is set to the amount of data actually read on return.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn	IN	X
ncb_num	IN	X
ncb_buffer	IN	X
ncb_length	IN/OUT	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBRECVANY

This command receives data from any session corresponding to the specified name. This command can also be used to receive data destined for any local name by setting the *ncb_num* field to 0xFF. Otherwise, simply set *ncb_num* to the network number returned from adding a name to the local name table. Then any data pending for that particular name will be picked up by this command. Also, a precedence order exists for when multiple receive commands are outstanding. See the entry for *NCBRECV* for more details.

When a session is closed by a local session close command, by the remote side closing the session, or by a session abort command, any outstanding *NRCRECVANY* commands for the specified name will complete with the error *NRC_SCLOSED* (0x0A); the *ncb_lsn* field of the *NCB* structure is set to the local session number that was terminated. If no *NCBRECVANY* commands for that closed session are pending for the specified name and an outstanding *NCBRECVANY* command exists for any session (*ncb_num* is 0xFF), that command will complete with the error *NRC_SCLOSED* and with the *ncb_lsn* field set to the corresponding session number.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn	OUT	
ncb_num	IN/OUT	X
ncb_buffer	IN	X
ncb_length	IN/OUT	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBRESET

This command resets the specified LANA number and affects certain environment resources.

- If *ncb_lsn* is not 0, all resources associated with *ncb_lana_num* are freed.
- If *ncb_lsn* is 0, all resources associated with *ncb_lana_num* are freed and new resources are allocated. The *ncb_callname[0]* byte specifies the maximum number of sessions, the *ncb_callname[2]* byte specifies the maximum number of names, and the *ncb_callname[3]* byte requests that the application use the computer's name (which has the name number 1).

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn	IN	X
ncb_num	IN	X
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post		
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event		

NCBSEND

This command sends data to the specified session partner. The maximum data size that can be transmitted is 65,536 bytes (64 KB). If the remote side issues a hang up command, all pending sends return the "session closed" error, *NRC_SCLOSED* (0x0A). If more than one send command is pending, they are processed in first-in, first-out order.

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn	IN	X
ncb_num		
ncb_buffer	IN	X
ncb_length	IN	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBSENDNA

This command sends data to a specified session and does not wait for acknowledgment from the session partner. Otherwise, the behavior of this command is the same as that of *NCBSEND*.

Field	*In/Out*	*Required*
ncb_command	IN	
ncb_retcode	OUT	
ncb_lsn	IN	X
ncb_num		
ncb_buffer	IN	X
ncb_length	IN	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBSSTAT

This command retrieves the status of a session. When calling this command, *ncb_buffer* is set to a block of memory that will be filled with a *SESSION_HEADER* structure followed by one or more *SESSION_BUFFER* structures. If the first byte of *ncb_name* is an asterisk (*), this command obtains the status for all sessions associated with all names in the local name table. If the supplied buffer is too small, the error *NRC_INCOMP* (0x06) is returned. If the buffer length is less than 4, the error returned is *NRC_BUFLEN* (0x01).

Field	*In/Out*	*Required*
ncb_command	IN	X
ncb_retcode	OUT	
ncb_lsn		
ncb_num	OUT	
ncb_buffer	IN	X
ncb_length	IN	X
ncb_callname		
ncb_name	IN	X
ncb_rto		
ncb_sto		
ncb_post	IN	
ncb_lana_num	IN	X
ncb_cmd_cplt	OUT	
ncb_event	IN	

NCBUNLINK

This command unlinks the adapter and is provided for compatibility with earlier versions of NetBIOS. It has no effect on Win32 platforms.

Appendix B -- IP Helper Functions

This appendix will introduce you to some new API functions that allow you to query and manage IP protocol characteristics on your computer. The functions are designed to help you programmatically achieve the functionality that is available in the following standard IP utilities:

- Ipconfig.exe (or Winipcfg.exe on Microsoft Windows 95) Displays IP configuration information and permits you to release and renew DHCP-assigned IP addresses.
- Netstat.exe Displays the TCP connection table, the UDP listener table, and the IP protocol statistics.
- Route.exe Displays and manipulates network routing tables.
- Arp.exe Displays and modifies the IP-to-physical address translation tables used by the address resolution protocol (ARP).

The functions described in this appendix are available mainly on Windows 98 and Windows 2000. Several are also available on Windows NT 4 Service Pack 4 or later; however, none are available on Windows 95. We will point out platform specifics as we discuss each function. The prototypes for all the functions described in this appendix are defined in Iphlpapi.h. When you build your application, you must link it to the library file Iphlpapi.lib.

Ipconfig

The Ipconfig.exe utility presents two pieces of information: IP configuration information and IP configuration parameters specific to each network adapter installed on your machine. To retrieve IP configuration information, use the *GetNetworkParams* function, which is defined as

```
DWORD GetNetworkParams(
    PFIXED_INFO pFixedInfo,
    PULONG pOutBufLen
);
```

The *pFixedInfo* parameter receives a pointer to a buffer that receives a *FIXED_INFO* data structure your application must provide to retrieve the IP configuration information. The *pOutBufLen* parameter is a pointer to a variable that specifies the size of the buffer you passed in the *pFixedInfo* parameter. If your buffer is not large enough, *GetNetworkParams* returns *ERROR_BUFFER_OVERFLOW* and sets the *pOutBufLen* parameter to the required buffer size.

The *FIXED_INFO* structure used in *GetNetworkParams* is defined as

```
typedef struct
{
    char            HostName[MAX_HOSTNAME_LEN + 4] ;
    char            DomainName[MAX_DOMAIN_NAME_LEN + 4];
    PIP_ADDR_STRING CurrentDnsServer;
    IP_ADDR_STRING  DnsServerList;
    UINT            NodeType;
    char            ScopeId[MAX_SCOPE_ID_LEN + 4];
    UINT            EnableRouting;
    UINT            EnableProxy;
    UINT            EnableDns;
} FIXED_INFO, *PFIXED_INFO;
```

The fields are defined as follows:

- *HostName* Represents your computer's name as recognized by DNS.
- *DomainName* Specifies the DNS domain your computer belongs to.
- *CurrentDnsServer* Contains the IP address of the current DNS server.
- *DnsServerList* Is a linked list containing the DNS servers that your machine uses.
- *NodeType* Specifies how the system resolves NetBIOS names over an IP network. Table B-1 contains the possible values.
- *ScopeId* Identifies a string value that is appended to a NetBIOS name to logically group two or more computers for NetBIOS communication over TCP/IP.
- *EnableRouting* Specifies whether the system will route IP packets between the networks it is connected to.
- *EnableProxy* Specifies whether the system will act as a WINS proxy agent on a network. A WINS proxy

agent answers broadcast queries for names that it has resolved through WINS and allows a network of b-node computers to connect to servers on other subnets registered with WINS.

- *EnableDns* Specifies whether NetBIOS will query DNS for names that cannot be resolved by WINS, broadcast, or the *LMHOSTS* file.

Table B-1. *Possible node type values*

Value	*Description*
BROADCAST_NODETYPE	Known as b-node NetBIOS name resolution, in which the system uses IP broadcasting to perform NetBIOS name registration and name resolution.
PEER_TO_PEER_NODETYPE	Known as p-node NetBIOS name resolution, in which the system uses point-to-point communication with a NetBIOS name server (such as WINS) to register and resolve computer names to IP addresses.
MIXED_NODETYPE	Known as m-node (mixed node) NetBIOS name resolution, in which the system uses both the b-node and p-node techniques just described. The b-node method is used first; if it fails, the p-node method is used next.
HYBRID_NODETYPE	Known as h-node (hybrid node) NetBIOS name resolution, in which the system uses both the b-node and p-node techniques. The p-node method is used first; if it fails, the b-node method is used next.

The *DnsServerList* field of a *FIXED_INFO* structure is an *IP_ADDR_STRING* structure that represents the beginning of a linked list of IP addresses. This field is defined as

```
typedef struct _IP_ADDR_STRING
{
    struct _IP_ADDR_STRING* Next;
    IP_ADDRESS_STRING       IpAddress;
    IP_MASK_STRING          IpMask;
    DWORD                   Context;
} IP_ADDR_STRING, *PIP_ADDR_STRING;
```

The *Next* field identifies the next DNS server IP address in the list. If *Next* is set to *NULL*, it indicates the end of the list. The *IpAddress* field is a string of characters that represents an IP address as a dotted decimal string. The *IpMask* field is a string of characters that represents the subnet mask associated with the IP address listed in *IpAddress*. The final field, *Context*, identifies the IP address with a unique value on the system.

The Ipconfig.exe utility is also capable of retrieving IP configuration information specific to a network interface. A network interface can be a hardware Ethernet adapter or even a RAS dial-up adapter. You can retrieve adapter information by calling *GetAdaptersInfo*, which is defined below.

```
DWORD GetAdaptersInfo (
    PIP_ADAPTER_INFO pAdapterInfo,
    PULONG pOutBufLen
);
```

Use the *pAdapterInfo* parameter to pass a pointer to an application-provided buffer that receives an *ADAPTER_INFO* data structure with the adapter configuration information. The *pOutBufLen* parameter is a pointer to a variable that specifies the size of the buffer you passed in the *pAdapterInfo* parameter. If your buffer is not large enough, *GetAdaptersInfo* returns *ERROR_BUFFER_OVERFLOW* and sets the *pOutBufLen* parameter to the required buffer size.

The *IP_ADAPTER_INFO* structure is actually a list of structures containing IP configuration information specific to every network adapter available on your machine. *IP_ADAPTER_INFO* is defined as

```
typedef struct _IP_ADAPTER_INFO
{
    struct _IP_ADAPTER_INFO* Next;
    DWORD           ComboIndex;
    char            AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
    char            Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
    UINT            AddressLength;
    BYTE            Address[MAX_ADAPTER_ADDRESS_LENGTH];
    DWORD           Index;
    UINT            Type;
    UINT            DhcpEnabled;
    PIP_ADDR_STRING CurrentIpAddress;
    IP_ADDR_STRING  IpAddressList;
    IP_ADDR_STRING  GatewayList;
    IP_ADDR_STRING  DhcpServer;
    BOOL            HaveWins;
    IP_ADDR_STRING  PrimaryWinsServer;
    IP_ADDR_STRING  SecondaryWinsServer;
    time_t          LeaseObtained;
    time_t          LeaseExpires;
} IP_ADAPTER_INFO, *PIP_ADAPTER_INFO;
```

The fields of the structure are defined as follows:

- *Next* Identifies the next adapter in the buffer. A *NULL* value indicates the end of the list.
- *ComboIndex* Is not used and will be set to 0.
- *AdapterName* Identifies the name of the adapter.
- *Description* Is a simple description of the adapter.
- *AddressLength* Identifies how many bytes make up the physical address of the adapter in the *Address* field.
- *Address* Identifies the physical address of the adapter.
- *Index* Identifies a unique internal index number of the network interface that this adapter is assigned to.
- *Type* Specifies the type of the adapter as a numeric value. Table B-2 defines the adapter types supported.

Table B-2. *Adapter types*

Adapter Type Value	*Description*
MIB_IF_TYPE_ETHERNET	Ethernet adapter
MIB_IF_TYPE_FDDI	FDDI adapter
MIB_IF_TYPE_LOOPBACK	Loopback adapter
MIB_IF_TYPE_OTHER	Other type of adapter
MIB_IF_TYPE_PPP	PPP adapter
MIB_IF_TYPE_SLIP	Slip adapter
MIB_IF_TYPE_TOKENRING	Token Ring adapter

- DhcpEnabled Specifies whether DHCP is enabled on this adapter.
- CurrentIpAddress Is not used and will be set to a NULL value.
- IpAddressList Specifies a list of IP addresses assigned to the adapter.
- GatewayList Specifies a list of IP addresses representing the default gateway.
- DhcpServer Specifies a list with only one element representing the IP address of the DHCP server used.
- HaveWins Specifies whether the adapter uses a WINS server.
- PrimaryWinsServer Specifies a list with only one element representing the IP address of the primary WINS server used.
- SecondaryWinsServer Specifies a list with only one element representing the IP address of the secondary WINS server used.
- LeaseObtained Identifies when the lease for the IP address was obtained from a DHCP server.
- LeaseExpires Identifies when the lease on the IP address obtained from DHCP expires.

Releasing and Renewing IP Addresses

The Ipconfig.exe utility also features the ability to release and renew IP addresses obtained from the DHCP server by specifying the /release and /renew command line parameters. If you want to programmatically release an IP address, you can call the *IPReleaseAddress* function, which is defined as

```
DWORD IpReleaseAddress (
    PIP_ADAPTER_INDEX_MAP AdapterInfo
);
```

If you want to renew an IP address, you can call the *IPRenewAddress* function, which is defined as

```
DWORD IpRenewAddress (
    PIP_ADAPTER_INDEX_MAP AdapterInfo
);
```

Each of these two functions features an *AdapterInfo* parameter that is an *IP_ADAPTER_INDEX_MAP* structure,

which identifies the adapter to release or renew the address for. The *IP_ADAPTER_INDEX_MAP* structure is defined as

```
typedef struct _IP_ADAPTER_INDEX_MAP
{
    ULONG Index;
    WCHAR Name[MAX_ADAPTER_NAME];
}IP_ADAPTER_INDEX_MAP, *PIP_ADAPTER_INDEX_MAP;
```

The fields of this structure are defined as follows:

- *Index* Identifies the internal index of the network interface that the adapter is assigned to.
- *Name* Identifies the name of the adapter.

You can retrieve the *IP_ADAPTER_INDEX_MAP* structure for a particular adapter by calling the *GetInterfaceInfo* function, which is defined as

```
DWORD GetInterfaceInfo (
    IN PIP_INTERFACE_INFO pIfTable,
    OUT PULONG dwOutBufLen
);
```

The *pIfTable* parameter is a pointer to an *IP_INTERFACE_INFO* application buffer that will receive interface information. The *dwOutBufLen* parameter is a pointer to a variable that specifies the size of the buffer you passed in the *pIfTable* parameter. If the buffer is not large enough to hold the interface information, *GetInterfaceInfo* returns the error *ERROR_INSUFFICIENT_BUFFER* and sets the *dwOutBufLen* parameter to the required buffer size.

The *IP_INTERFACE_INFO* structure is defined as

```
typedef struct _IP_INTERFACE_INFO
{
    LONG                NumAdapters;
    IP_ADAPTER_INDEX_MAP Adapter[1];
} IP_INTERFACE_INFO,*PIP_INTERFACE_INFO;
```

Its fields are defined as follows:

- *NumAdapters* Identifies the number of adapters in the *Adapter* field.
- *Adapter* Is an array of *IP_ADAPTER_INDEX_MAP* structures, defined above.

Once you have obtained the *IP_ADAPTER_INDEX_MAP* structure for a particular adapter, you can release or renew the DHCP-assigned IP address using the *IPReleaseAddress* and *IPRenewAddress* functions we just described.

Changing IP Addresses

The Ipconfig.exe utility does not allow you to change an IP address for a network adapter (except in the case of

DHCP). However, two functions will allow you to add or delete an IP address for a particular adapter: the *AddIpAddress* and *DeleteIpAddress* IP Helper functions. These functions require you to understand adapter index numbers and IP context numbers. In Windows, every network adapter has a unique index ID (which we described earlier), and every IP address has a unique context ID. Adapter index IDs and IP context numbers can be retrieved using *GetAdaptersInfo*. The *AddIpAddress* function is defined as

```
DWORD AddIPAddress (
    IPAddr Address,
    IPMask IpMask,
    DWORD IfIndex,
    PULONG NTEContext,
    PULONG NTEInstance
);
```

The *Address* parameter specifies the IP address to add as an unsigned long value. The *IpMask* parameter specifies the subnet mask for the IP address as an unsigned long value. The *IfIndex* parameter specifies the adapter index to add the address to. The *NTEContext* parameter receives the context value associated with the IP address added. The *NTEInstance* parameter receives an instance value associated with an IP address.

If you want to programmatically delete an IP address for an adapter, you can call *DeleteIpAddress*, which is defined below.

```
DWORD DeleteIPAddress (
    ULONG NTEContext
);
```

The *NTEContext* parameter identifies a context value associated with an IP address. This value can be obtained from *GetAdaptersInfo*, which we described earlier in the chapter.

Netstat

The Netstat.exe utility displays the TCP connection table, the UDP listener table, and the IP protocol statistics on your computer. The functions used to retrieve this information not only work on Windows 98 and Windows 2000 but are also available on Windows NT 4 Service Pack 4 (or later).

Retrieving the TCP Connection Table

The *GetTcpTable* function retrieves the TCP connection table. This is the same information you see when you execute Netstat.exe with the -p tcp -a options. *GetTcpTable* is defined as

```
DWORD GetTcpTable(
    PMIB_TCPTABLE pTcpTable,
    PDWORD pdwSize,
    BOOL bOrder
);
```

The *pTcpTable* parameter is a pointer to an *MIB_TCPTABLE* application buffer that will receive the TCP connection information. The *pdwSize* parameter is a pointer to a variable that specifies the size of the buffer you passed in the *pTcpTable* parameter. If the buffer is not large enough to hold the TCP information, the function sets this parameter to the required buffer size. The *bOrder* parameter specifies whether the returned information should be sorted.

The *MIB_TCPTABLE* structure returned from *GetTcpTable* is defined as

```
typedef struct _MIB_TCPTABLE
{
    DWORD dwNumEntries;
    MIB_TCPROW table[ANY_SIZE];
} MIB_TCPTABLE, *PMIB_TCPTABLE;
```

The fields of this structure are defined as follows:

- *dwNumEntries* Specifies how many entries are in the *table* field (described next).
- *table* Is a pointer to an array of *MIB_TCPROW* structures that contain TCP connection information.

The *MIB_TCPROW* structure contains the IP address pair that comprises a TCP connection. This structure is defined as

```
typedef struct _MIB_TCPROW
{
    DWORD dwState;
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
    DWORD dwRemoteAddr;
    DWORD dwRemotePort;
} MIB_TCPROW, *PMIB_TCPROW;
```

Its fields are defined as follows:

- *dwState* Specifies the state of the TCP connection, as defined in Table B-3.

Table B-3. *TCP connection states*

Connection State	*RFC 793 Description*
MIB_TCP_STATE_CLOSED	Known as the "CLOSED" state
MIB_TCP_STATE_CLOSING	Known as the "CLOSING" state
MIB_TCP_STATE_CLOSE_WAIT	Known as the "CLOSE WAIT" state
MIB_TCP_STATE_DELETE_TCB	Known as the "DELETE" state
MIB_TCP_STATE_ESTAB	Known as the "ESTABLISHED" state
MIB_TCP_STATE_FIN_WAIT1	Known as the "FIN WAIT1" state
MIB_TCP_STATE_FIN_WAIT2	Known as the "FIN WAIT2" state
MIB_TCP_STATE_LAST_ACK	Known as the "LAST ACK" state
MIB_TCP_STATE_LISTEN	Known as the "LISTENING" state
MIB_TCP_STATE_SYN_RCVD	Known as the "SYN RCVD" state
MIB_TCP_STATE_SYN_SENT	Known as the "SYN SENT" state
MIB_TCP_STATE_TIME_WAIT	Known as the "TIME WAIT" state

- *dwLocalAddr* Specifies a local IP address for the connection.
- *dwLocalPort* Specifies a local port for the connection.
- *dwRemoteAddr* Specifies the remote IP address for the connection.
- *dwRemotePort* Specifies the remote port for the connection.

Retrieving the UDP Listener Table

The *GetUdpTable* function retrieves the UDP listener table. This is the same information you see if you execute Netstat.exe with the -p udp -a options. *GetUdpTable* is defined as

```
DWORD GetUdpTable(
    PMIB_UDPTABLE pUdpTable,
    PDWORD pdwSize,
    BOOL bOrder
);
```

The *pUdpTable* parameter is a pointer to an *MIB_UDPTABLE* application buffer that will receive the UDP listener information. The *pdwSize* parameter is a pointer to a variable that specifies the size of the buffer you passed in the *pUdpTable* parameter. If the buffer is not large enough to hold the UDP information, the function sets this parameter to the required buffer size. The *bOrder* parameter specifies whether the returned information should be sorted.

The *MIB_UDPTABLE* structure returned from *GetUdpTable* is defined as

```
typedef struct _MIB_UDPTABLE
{
    DWORD dwNumEntries;
    MIB_UDPROW table[ANY_SIZE];
} MIB_UDPTABLE, * PMIB_UDPTABLE;
```

The fields of this structure are defined as follows:

- *dwNumEntries* Specifies how many entries are in the *table* field, described next.
- *table* Is a pointer to an array of *MIB_UDPROW* structures that contain UDP listener information.

The *MIB_UDPROW* structure contains the IP address in which UDP is listening for datagrams. This structure is defined as

```
typedef struct _MIB_UDPROW
{
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
} MIB_UDPROW, * PMIB_UDPROW;
```

Its fields are defined as follows:

- *dwLocalAddr* Specifies the local IP address.
- *dwLocalPort* Specifies the local IP port.

Retrieving IP Protocol Statistics

Four functions are available for receiving IP statistics: *GetIpStatistics*, *GetIcmpStatistics*, *GetTcpStatistics*, and *GetUdpStatistics*. These functions produce the same information that is returned from Netstat.exe when you call it with the -s parameter. The first statistics function, *GetIpStatistics*, retrieves the IP statistics for the current computer and is defined as

```
DWORD GetIpStatistics(
    PMIB_IPSTATS pStats
);
```

The *pStats* parameter is a pointer to an *MIB_IPSTATS* structure that receives the current IP statistics for your computer. The *MIB_IPSTATS* structure is defined as

```
typedef struct _MIB_IPSTATS
{
    DWORD dwForwarding;
    DWORD dwDefaultTTL;
    DWORD dwInReceives;
    DWORD dwInHdrErrors;
    DWORD dwInAddrErrors;
    DWORD dwForwDatagrams;
    DWORD dwInUnknownProtos;
    DWORD dwInDiscards;
    DWORD dwInDelivers;
    DWORD dwOutRequests;
    DWORD dwRoutingDiscards;
    DWORD dwOutDiscards;
    DWORD dwOutNoRoutes;
    DWORD dwReasmTimeout;
    DWORD dwReasmReqds;
    DWORD dwReasmOks;
    DWORD dwReasmFails;
    DWORD dwFragOks;
    DWORD dwFragFails;
    DWORD dwFragCreates;
    DWORD dwNumIf;
    DWORD dwNumAddr;
    DWORD dwNumRoutes;
} MIB_IPSTATS, *PMIB_IPSTATS;
```

The fields of this structure are defined as follows:

- *dwForwarding* Specifies whether IP forwarding is enabled or disabled on your computer.
- *dwDefaultTTL* Specifies the initial time-to-live (TTL) value for datagrams originating on your computer.
- *dwInReceives* Specifies the number of datagrams received.
- *dwInHdrErrors* Specifies the number of datagrams received with bad headers.
- *dwInAddrErrors* Specifies the number of datagrams received with bad addresses.
- *dwForwDatagrams* Specifies the number of datagrams forwarded.
- *dwInUnknownProtos* Specifies the number of datagrams received with an unknown protocol.
- *dwInDiscards* Specifies the number of datagrams received that were discarded.

- *dwInDelivers* Specifies the number of datagrams received that were delivered.
- *dwOutRequests* Specifies the number of datagrams that IP has requested to transmit.
- *dwRoutingDiscards* Specifies the number of outgoing datagrams discarded.
- *dwOutDiscards* Specifies the number of transmitted datagrams discarded.
- *dwOutNoRoutes* Specifies the number of datagrams that did not have a routing destination.
- *dwReasmTimeout* Specifies the maximum amount of time for a fragmented datagram to arrive.
- *dwReasmReqds* Specifies the number of datagrams that require assembly.
- *dwReasmOks* Specifies the number of datagrams that were successfully reassembled.
- *dwFragFails* Specifies the number of datagrams that could not be fragmented.
- *dwFragCreates* Specifies the number of datagrams that were fragmented.
- *dwNumIf* Specifies the number of IP interfaces available on your computer.
- *dwNumAddr* Specifies the number of IP addresses identified on your computer.
- *dwNumRoutes* Specifies the number of routes available in the routing table.

The second statistics function, *GetIcmpStatistics*, retrieves Internet Control Message Protocol (ICMP) statistics and is defined as

```
DWORD GetIcmpStatistics(
   PMIB_ICMP pStats
);
```

The *pStats* parameter is a pointer to an *MIB_ICMP* structure that receives the current ICMP statistics for your computer. The *MIB_ICMP* structure is defined as

```
typedef struct _MIB_ICMP
{
    MIBICMPINFO stats;
} MIB_ICMP,*PMIB_ICMP;
```

As you can see, *MIB_ICMP* is a structure containing a *MIBICMPINFO* structure that is defined as

```
typedef struct _MIBICMPINFO
{
    MIBICMPSTATS icmpInStats;
    MIBICMPSTATS icmpOutStats;
} MIBICMPINFO;
```

The *MIBICMPINFO* structure receives incoming or outgoing ICMP information through an *MIBICMPSTATS* structure. The *icmpInStats* parameter receives incoming data, while *icmpOutStats* receives outgoing data. The

MIBICMPSTATS structure is defined as

```
typedef struct _MIBICMPSTATS
{
    DWORD dwMsgs;
    DWORD dwErrors;
    DWORD dwDestUnreachs;
    DWORD dwTimeExcds;
    DWORD dwParmProbs;
    DWORD dwSrcQuenchs;
    DWORD dwRedirects;
    DWORD dwEchos;
    DWORD dwEchoReps;
    DWORD dwTimestamps;
    DWORD dwTimestampReps;
    DWORD dwAddrMasks;
    DWORD dwAddrMaskReps;
} MIBICMPSTATS;
```

The fields of this structure are defined as follows:

- *dwMsgs* Specifies the number of messages sent or received.
- *dwErrors* Specifies the number of errors sent or received.
- *dwDestUnreachs* Specifies the number of "destination unreachable" messages sent or received.
- *dwTimeExcds* Specifies the number of TTL-exceeded messages sent or received.
- *dwParmProbs* Specifies the number of messages sent or received that indicate a datagram contains bad IP information.
- *dwSrcQuenchs* Specifies the number of source quench messages sent or received.
- *dwRedirects* Specifies the number of redirection messages sent or received.
- *dwEchos* Specifies the number of ICMP echo requests sent or received.
- *dwEchoReps* Specifies the number of ICMP echo replies sent or received.
- *dwTimestamps* Specifies the number of timestamp requests sent or received.
- *dwTimestampReps* Specifies the number of timestamp replies sent or received.
- *dwAddrMasks* Specifies the number of address masks sent or received.
- *dwAddrMaskReps* Specifies the number of address mask replies sent or received.

The third statistics function, *GetTcpStatistics*, retrieves TCP statistics on your computer and is defined as

```
DWORD GetTcpStatistics(
    PMIB_TCPSTATS pStats
);
```

The *pStats* parameter is a pointer to an *MIB_TCPSTATS* structure that receives the current IP statistics for your computer. The *MIB_TCPSTATS* structure is defined as

```
typedef struct _MIB_TCPSTATS
{
    DWORD dwRtoAlgorithm;
    DWORD dwRtoMin;
    DWORD dwRtoMax;
    DWORD dwMaxConn;
    DWORD dwActiveOpens;
    DWORD dwPassiveOpens;
    DWORD dwAttemptFails;
    DWORD dwEstabResets;
    DWORD dwCurrEstab;
    DWORD dwInSegs;
    DWORD dwOutSegs;
    DWORD dwRetransSegs;
    DWORD dwInErrs;
    DWORD dwOutRsts;
    DWORD dwNumConns;
} MIB_TCPSTATS, *PMIB_TCPSTATS;
```

The fields of this structure are defined as follows:

- *dwRtoAlgorithm* Specifies which retransmission algorithm is being used. The valid values are *MIB_TCP_RTO_CONSTANT*, *MIB_TCP_RTO_RSRE*, *MIB_TCP_RTO_VANJ*, and *MIB_TCP_RTO_OTHER*, which is for other types.
- *dwRtoMin* Specifies the minimum retransmission timeout in milliseconds.
- *dwRtoMax* Specifies the maximum retransmission timeout in milliseconds.
- *dwMaxConn* Specifies the maximum number of connections allowed.
- *dwActiveOpens* Specifies how many times the machine is initiating a connection with a server.
- *dwPassiveOpens* Specifies how many times the machine is listening for a connection from a client.
- *dwAttemptFails* Specifies how many connection attempts have failed.
- *dwEstabResets* Specifies the number of established connections that have been reset.
- *dwCurrEstab* Specifies the number of connections that are currently established.
- *dwInSegs* Specifies the number of segments received.
- *dwOutSegs* Specifies the number of segments transmitted (excluding segments that have been retransmitted).
- *dwRetransSegs* Specifies the number of segments retransmitted.
- *dwInErrs* Specifies the number of errors received.
- *dwOutRsts* Specifies the number of segments transmitted with the reset flag set.
- *dwNumConns* Specifies the total number of connections.

The last statistics function, *GetUdpStatistics*, retrieves UDP statistics on your computer and is defined as

```
DWORD GetUdpStatistics(
    PMIB_UDPSTATS pStats
);
```

The *pStats* parameter is a pointer to an *MIB_UDPSTATS* structure that receives the current IP statistics for your computer. The *MIB_UDPSTATS* structure is defined as

```
typedef struct _MIB_UDPSTATS
{
    DWORD dwInDatagrams;
    DWORD dwNoPorts;
    DWORD dwInErrors;
    DWORD dwOutDatagrams;
    DWORD dwNumAddrs;
} MIB_UDPSTATS,*PMIB_UDPSTATS;
```

This structure's fields are defined as follows:

- *dwInDatagrams* Specifies the number of datagrams received.
- *dwNoPorts* Specifies the number of datagrams discarded because the port number was bad.
- *dwInErrors* Specifies the number of erroneous datagrams received (excluding the datagrams counted in *dwNoPorts*).
- *dwOutDatagrams* Specifies the number of datagrams transmitted.
- *dwNumAddrs* Specifies the total number of UDP entries in the listener table.

Route

The Route.exe command allows you to print and modify the routing table. The routing table determines which IP interface a connection request or a datagram occurs on. The IP Helper library offers several functions for manipulating the routing table. All of the functions related to routing are available on Windows 98, Windows 2000, and Windows NT 4 Service Pack 4 (or later).

First, let's discuss the capabilities of the Route.exe command. Its most basic function is to print the routing table. A route consists of a destination address, a netmask, a gateway, a local IP interface, and a metric. You also have the ability to add and delete a route. To add a route, you must specify all the parameters we just described. To delete a route, you must specify only the destination address. In this section, we'll look at the IP Helper functions that print the routing table. Then we'll discuss adding and deleting a route.

Getting the Routing Table

The most basic action that Route.exe performs is printing the table. This is accomplished with the *GetIpForwardTable* function, which is defined as

```
DWORD GetIpForwardTable (
    PMIB_IPFORWARDTABLE pIpForwardTable,
    PULONG pdwSize,
    BOOL bOrder
);
```

The first parameter, *pIpForwardTable*, contains the routing table information upon return. When you call the function, this parameter should point to a buffer of sufficient size. If you call the function with *pIpForwardTable* equal to *NULL* (or if the buffer size is insufficient to begin with), the *pdwSize* parameter returns the length of the buffer needed for the call to complete successfully. The last parameter, *bOrder*, indicates whether the results should be sorted upon return. The default sorting order is

1. Destination address
2. Protocol that generated the route
3. Multipath routing policy
4. Next-hop address

The routing information is returned in the form of the *MIB_IPFORWARDTABLE* structure, which is defined as

```
typedef struct _MIB_IPFORWARDTABLE
{
    DWORD            dwNumEntries;
    MIB_IPFORWARDROW table[ANY_SIZE];
} MIB_IPFORWARDTABLE, *PMIB_IPFORWARDTABLE;
```

This structure is a wrapper for an array of *MIB_IPFORWARDROW* structures. The *dwNumEntries* field indicates the number of these structures present in the array. The *MIB_IPFORWARDROW* structure is defined below.

```
typedef struct _MIB_IPFORWARDROW
{
    DWORD dwForwardDest;
    DWORD dwForwardMask;
    DWORD dwForwardPolicy;
    DWORD dwForwardNextHop;
    DWORD dwForwardIfIndex;
    DWORD dwForwardType;
    DWORD dwForwardProto;
    DWORD dwForwardAge;
    DWORD dwForwardNextHopAS;
    DWORD dwForwardMetric1;
    DWORD dwForwardMetric2;
    DWORD dwForwardMetric3;
    DWORD dwForwardMetric4;
    DWORD dwForwardMetric5;
} MIB_IPFORWARDROW, *PMIB_IPFORWARDROW;
```

The fields of this structure are defined as follows:

- *dwForwardDest* Is the IP address of the destination host.
- *dwForwardMask* Is the subnet mask for the destination host.
- *dwForwardPolicy* Specifies a set of conditions that would cause the selection of a multipath route. Usually these conditions are in the form of IP Type of Service (TOS). For more information about TOS, see Chapter 9 and the *IP_TOS* option. For more information on multipath routing, see RFC 1354.
- *dwForwardNextHop* Is the IP address for the next hop in the route.
- *dwForwardIfIndex* Indicates the index of the interface for this route.
- *dwForwardType* Indicates the route type as defined in RFC 1354. Table B-4 lists the possible values and meanings for this field.
- *dwForwardProto* Is the protocol that generated the route. Table B-5 lists the possible values for this field. The values for IPX protocols are defined in Routprot.h, while the IP entries are included in Iprtrmib.h.
- *dwForwardAge* Indicates the age of the route in seconds.
- *dwForwardNextHopAS* Is the autonomous system number of the next hop.
- *dwForwardMetric1* Is a routing protocol-specific metric value. For more information, see RFC 1354. The field contains the route metric value that you normally see when executing the Route.exe print command. For this and the following four fields, if the entry is unused, the value is *MIB_IPROUTE_METRIC_UNUSED* (-1).
- *dwForwardMetric2* Is a routing protocol-specific metric value. For more information, see RFC 1354.
- *dwForwardMetric3* Is a routing protocol-specific metric value. For more information, see RFC 1354.
- *dwForwardMetric4* Is a routing protocol-specific metric value. For more information, see RFC 1354.
- *dwForwardMetric5* Is a routing protocol-specific metric value. For more information, see RFC 1354.

Table B-4. *Possible route types for a routing table entry*

Forward Type	*Description*
MIB_IPROUTE_TYPE_INDIRECT	Next hop is not the final destination (remote route)
MIB_IPROUTE_TYPE_DIRECT	Next hop is the final destination (local route)
MIB_IPROUTE_TYPE_INVALID	Route is invalid
MIB_IPROUTE_TYPE_OTHER	Other route

Table B-5. *Routing protocol identifiers*

Protocol Identifier	*Description*
MIB_IPPROTO_OTHER	Protocol not listed
MIB_IPPROTO_LOCAL	Route generated by the stack
MIB_IPPROTO_NETMGMT	Route added by Route.exe utility or SNMP
MIB_IPPROTO_ICMP	Route from ICMP redirects
MIB_IPPROTO_EGP	Exterior Gateway Protocol
MIB_IPPROTO_GGP	Gateway Gateway Protocol
MIB_IPPROTO_HELLO	HELLO routing protocol
MIB_IPPROTO_RIP	Routing Information Protocol
MIB_IPPROTO_IS_IS	IP Intermediate System to Intermediate System Protocol
MIB_IPPROTO_ES_IS	IP End System to Intermediate System Protocol
MIB_IPPROTO_CISCO	IP Cisco protocol
MIB_IPPROTO_BBN	BBN protocol
MIB_IPPROTO_OSPF	Open Shortest Path First routing protocol
MIB_IPPROTO_BGP	Border Gateway Protocol
MIB_IPPROTO_NT_AUTOSTATIC	Routes that were originally added by a routing protocol but are not static
MIB_IPPROTO_NT_STATIC	Routes added by the routing user interface or the Routemon.exe utility
MIB_IPPROTO_STATIC_NON_DOD	Identical to *PROTO_IP_NT_STATIC* except that these routes will not cause Dial on Demand (DOD)
IPX_PROTOCOL_RIP	Routing Information Protocol for IPX
IPX_PROTOCOL_SAP	Service Advertisement Protocol
IPX_PROTOCOL_NLSP	Netware Link Services Protocol

Adding a Route

The next function of the route command is adding a route. Remember that to add a route, the destination IP, netmask, gateway, local IP interface, and metric must be specified. When adding a route, you should verify that the given local IP interface is valid. In addition, when adding a route you will need to refer to the local IP interface on which the route is based by its internal index value. You can obtain this information by calling the *GetIpAddrTable* function, which is defined as

```
DWORD GetIpAddrTable (
    PMIB_IPADDRTABLE pIpAddrTable,
    PULONG pdwSize,
    BOOL bOrder
);
```

The first parameter, *pIpAddrTable*, is a buffer of sufficient size that will return an *MIB_IPADDRTABLE* structure. The *pdwSize* parameter is the size of the buffer passed as the first parameter. The last parameter, *bOrder*, specifies whether to return the local IP interfaces by ascending IP addresses. To find out the required buffer size, you can pass *NULL* for *pIpAddrTable*. Upon return, *pdwSize* will indicate the required buffer size. The *MIB_IPADDRTABLE* structure is defined as

```
typedef struct _MIB_IPADDRTABLE
{
    DWORD dwNumEntries
    MIB_IPADDRROW table[ANY_SIZE];
} MIB_IPADDRTABLE, *PMIB_IPADDRTABLE;
```

This structure is a wrapper for an array of *MIB_IPADDRROW* structures. The *dwNumEntries* field indicates how many *MIB_IPADDRROW* entries are present in the *table* field array. The *MIB_IPADDRROW* structure is defined as

```
typedef struct _MIB_IPADDRROW
{
    DWORD    dwAddr;
    DWORD    dwIndex;
    DWORD    dwMask;
    DWORD    dwBCastAddr;
    DWORD    dwReasmSize;
    unsigned short unused1;
    unsigned short unused2;
} MIB_IPADDRROW, *PMIB_IPADDRROW;
```

The fields of this structure are defined as follows:

- *dwAddr* Is the IP address for a given interface.
- *dwIndex* Is the index of the interface associated with the IP address.
- *dwMask* Is the subnet mask for the IP address.
- *dwBCastAddr* Is the broadcast address.
- *dwReasmSize* Is the maximum reassembly size for datagrams received.
- *unused1* and *unused2* Are not currently used.

Using this function, you can verify that the local IP interface for the given route is valid. The function for adding the route to the routing table is *SetIpForwardEntry*, which is defined as

```
DWORD SetIpForwardEntry (
    PMIB_IPFORWARDROW pRoute
);
```

The only parameter is *pRoute*, which is a pointer to an *MIB_IPFORWARDROW* structure. This structure defines the elements needed to establish a new route. We have already discussed this structure and its member fields. To add a route, the values must be specified for the fields *dwForwardIfIndex*, *dwForwardDest*, *dwForwardMask*, *dwForwardNextHop*, and *dwForwardPolicy*.

Deleting a Route

The last action for the route utility is deleting a route, which is the easiest command to implement. When invoking the route command to delete a route, you must specify the destination address to delete. Then you can search for the *MIB_IPFORWARDROW* structure returned from *GetIpForwardTable* that corresponds to the destination address. The *MIB_IPFORWARDROW* structure can then be passed to the *DeleteIpForwardEntry* function to remove the given entry. This function is defined as

```
DWORD DeleteIpForwardEntry (
    PMIB_IPFORWARDROW pRoute
);
```

Alternatively, you can specify the fields of *pRoute* yourself. The fields that are required to remove a route are *dwForwardIfIndex*, *dwForwardDest*, *dwForwardMask*, *dwForwardNextHop*, and *dwForwardPolicy*.

ARP

The Arp.exe utility is used to view and manipulate the ARP cache. The Platform SDK sample that emulates Arp.exe by using the IP Helper functions is named Iparp.exe. ARP (which, as you'll recall, stands for address resolution protocol) is responsible for resolving an IP address to a physical MAC address. Machines cache this information for performance reasons, and it is possible to access this information through the Arp.exe utility. Using this utility, you can display the ARP table with the -a option, delete an entry with the -d option, or add an entry with the -s option. In the next section, we will describe how to print the ARP cache, add an entry to the ARP table, and delete ARP entries.

All the IP Helper functions discussed in this section are available on Windows 98, Windows 2000, and Windows NT 4 Service Pack 4 (or later).

The simplest function is obtaining the ARP table. The IP Helper function that obtains this table is *GetIpNetTable*, which is defined as

```
DWORD GetIpNetTable (
    PMIB_IPNETTABLE pIpNetTable,
    PULONG          pdwSize,
    BOOL            bOrder
);
```

The first parameter, *pIpNetTable*, is a pointer to an *MIB_IPNETTABLE* structure that returns the ARP information. You must supply a sufficiently large buffer when calling this function. As with most other IP Helper functions, passing *NULL* for this parameter will return the buffer size needed as the parameter *pdwSize* and the error *ERROR_INSUFFICIENT_BUFFER*. Otherwise, *pdwSize* indicates the size of the buffer passed as *pIpNetTable*. The last parameter, *bOrder*, indicates whether the returned IP entries should be sorted in ascending IP order.

The *MIB_IPNETTABLE* structure is a wrapper for an array of *MIB_IPNETROW* structures and is defined as

```
typedef struct _MIB_IPNETTABLE
{
    DWORD        dwNumEntries;
    MIB_IPNETROW table[ANY_SIZE];
} MIB_IPNETTABLE, *PMIB_IPNETTABLE;
```

The *dwNumEntries* field indicates the number of array entries present in the *table* field. The *MIB_IPNETROW* structure contains the actual ARP entry information and is defined as

```
typedef struct _MIB_IPNETROW {
    DWORD dwIndex;
    DWORD dwPhysAddrLen;
    BYTE  bPhysAddr[MAXLEN_PHYSADDR];
    DWORD dwAddr;
    DWORD dwType;
} MIB_IPNETROW, *PMIB_IPNETROW;
```

The fields of this structure are as follows:

- *dwIndex* Specifies the index of the adapter.
- *dwPhysAddrLen* Indicates the length, in bytes, of the physical address contained in the *bPhysAddr* field.
- *bPhysAddr* Is an array of bytes that contains the physical (MAC) address of the adapter.
- *dwAddr* Specifies the IP address of the adapter.
- *dwType* Indicates the type of the ARP entry. Table B-6 shows the possible values for this field.

Table B-6. *Possible ARP entry types*

ARP Type	*Meaning*
MIB_IPNET_TYPE_STATIC	Static entry
MIB_IPNET_TYPE_DYNAMIC	Dynamic entry
MIB_IPNET_TYPE_INVALID	Invalid entry
MIB_IPNET_TYPE_OTHER	Other entry

Adding an ARP Entry

The next function of ARP is adding an entry to the ARP cache, which is another relatively simple operation. The IP Helper function to add an ARP entry is *SetIpNetEntry* and is defined as

```
DWORD SetIpNetEntry (
    PMIB_IPNETROW pArpEntry
);
```

The only argument is the *MIB_IPNETROW* structure, which we covered in the previous section. To add an ARP entry, simply fill in the structure with the new ARP information. First you need to set the *dwIndex* field to the index of a local IP address that indicates the network on which the ARP entry applies. Remember that if you are given the IP address, you can map the IP to the index with the *GetIpAddrTable* function. The next field, *dwPhysAddrLen*, is typically set to 6. (Most physical addresses, such as ETHERNET MAC addresses, are 6 bytes in length.) The *bPhysAddr* byte array must be set to the physical address. Most MAC addresses are represented as 12 characters—for example, 00-A0-C9-A7-86-E8. These characters need to be encoded into the proper byte array locations of the *bPhysAddr* field. For example, the sample MAC address would be encoded into the following bytes:

```
00000000 10100000 11001001 10100111 10000110 11101000
```

The encoding method is the same used for encoding IPX and ATM addresses. (See Chapter 6 for more information.) The *dwAddr* field must be set to the IP address of the remote host you are entering the MAC address of. The last field, *dwType*, is set to one of the ARP entry types listed in Table B-6. Once the structure is filled, call *SetIpNetEntry* to add the ARP entry to the cache. Upon success, *NO_ERROR* is returned.

Deleting an ARP Entry

Deleting an ARP entry is similar to adding one except that the only information required is the interface index, *dwIndex*, and the IP address of the ARP entry to delete, *dwAddr*. The function to remove an ARP entry is *DeleteIpNetEntry*, which is defined as

```
DWORD DeleteIpNetEntry (
    PMIB_IPNETROW pArpEntry
);
```

Again, the only parameter is an *MIB_IPNETROW* structure, and the only information necessary for removing an ARP entry is the local IP index and the IP address of the entry to delete. Remember that the index number to a local IP interface can be obtained with the function *GetIpAddrTable*. Upon success, *NO_ERROR* is returned.

Appendix C -- Winsock Error Codes

This appendix lists the Winsock error codes by error number. This list does not include the Winsock errors marked BSD-specific or undocumented. In addition, the Winsock errors that map directly to Win32 errors appear toward the end of this appendix.

10004—*WSAEINTR*

Interrupted function call. This error indicates that a blocking call was interrupted by a call to *WSACancelBlockingCall*.

10009—*WSAEBADF*

Bad file handle. This error means that the supplied file handle is invalid. Under Microsoft Windows CE, it is possible for the *socket* function to return this error, which indicates that the shared serial port is busy.

10013—*WSAEACCES*

Permission denied. An attempt was made to manipulate the socket, which is forbidden. This error most commonly occurs when attempting to use a broadcast address in *sendto* or *WSASendTo*, in which broadcast permission has not been set with *setsockopt* and the *SO_BROADCAST* option.

10014—*WSAEFAULT*

Invalid address. The pointer address passed into the Winsock function is invalid. This error is also generated when the specified buffer is too small.

10022—*WSAEINVAL*

Invalid argument. An invalid argument was specified. For example, specifying an invalid control code to *WSAIoctl* will generate this error. Plus, it can indicate an error with the current state of a socket—for example, calling *accept* or *WSAAccept* on a socket that is not listening.

10024—*WSAEMFILE*

Too many open files. Too many sockets are open. Typically, the Microsoft providers are limited only by the amount of resources available on the system.

10035—*WSAEWOULDBLOCK*

Resource temporarily unavailable. This error is most commonly returned on nonblocking sockets in which the requested operation cannot complete immediately. For example, calling *connect* on a nonblocking socket will return this error because the connection request cannot be completed immediately.

10036—*WSAEINPROGRESS*

Operation now in progress. A blocking operation is currently executing. Typically, you will not see this error unless you are developing 16-bit Winsock applications.

10037—*WSAEALREADY*

Operation already in progress. This error typically occurs when an operation that is already in progress is attempted on a nonblocking socket—for example, calling *connect* or *WSAConnect* a second time on a nonblocking socket already in the process of connecting. This error can also occur when a service provider is in the process of executing a callback function (for those Winsock functions that support callback routines).

10038—*WSAENOTSOCK*

Socket operation on an invalid socket. This error can be returned from any Winsock function that takes a *SOCKET* handle as a parameter. This error indicates that the supplied socket handle is not valid.

10039—*WSAEDESTADDRREQ*

Destination address required. This error indicates that the supplied address was omitted. For instance, calling

sendto with the destination address *INADDR_ANY* will return this error.

10040—*WSAEMSGSIZE*

Message too long. This error can mean a number of things. If a message is sent on a datagram socket that is too large for the internal buffer, this error occurs. It also occurs if the message is too large because of a network limitation. Finally, if upon receiving a datagram the buffer is too small to receive the message, this error is generated.

10041—*WSAEPROTOTYPE*

Wrong protocol type for socket. A protocol was specified in a call to *socket* or *WSASocket* that does not support the semantics of the given socket type. For example, requesting to create an IP socket of type *SOCK_STREAM* and protocol *IPPROTO_UDP* will generate this error.

10042—*WSAENOPROTOOPT*

Bad protocol option. An unknown, unsupported, or invalid socket option or level was specified in a call to *getsockopt* or *setsockopt*.

10043—*WSAEPROTONOSUPPORT*

Protocol not supported. Either the requested protocol is not installed on the system or no implementation exists for it. For example, if TCP/IP is not installed on the system, attempting to create either a TCP or a UDP socket will generate this error.

10044—*WSAESOCKTNOSUPPORT*

Socket type not supported. Support for the specified socket type does not exist for the given address family. For example, requesting a socket of type *SOCK_RAW* for a protocol that does not support raw sockets will generate this error.

10045—*WSAEOPNOTSUPP*

Operation not supported. The attempted operation is not supported for the referenced object. Typically, this occurs when trying to call a Winsock function on a socket that does not support that operation. For example, calling *accept* or *WSAAccept* on a datagram socket will cause this error.

10046—*WSAEPFNOSUPPORT*

Protocol family not supported. The requested protocol family does not exist or is not installed on the system. In most cases, this error is interchangeable with *WSAEAFNOSUPPORT*, which occurs more often.

10047—*WSAEAFNOSUPPORT*

Address family does not support requested operation. This error occurs when attempting to perform an operation that is not supported by the socket type. For example, trying to call *sendto* or *WSASendTo* with a socket of type *SOCK_STREAM* will generate this error. This error can also occur when calling *socket* or *WSASocket* and requesting an invalid combination of address family, socket type, and protocol.

10048—*WSAEADDRINUSE*

Address already in use. Under normal circumstances, only one socket is permitted to use each socket address. (For example, an IP socket address consists of the local IP address and port number.) This error is usually associated with the *bind*, *connect*, and *WSAConnect* functions. The socket option *SO_REUSEADDR* can be set with the *setsockopt* function to allow multiple sockets access to the same local IP address and port. (For more information, see Chapter 9.)

10049—*WSAEADDRNOTAVAIL*

Cannot assign requested address. This error occurs when the address specified in an API call is not valid for that function. For example, specifying an IP address in *bind* that does not correspond to a local IP interface will generate this error. This error also can occur when specifying port 0 for the remote machine to connect to with *connect*, *WSAConnect*, *sendto*, *WSASendTo*, and *WSAJoinLeaf*.

10050—*WSAENETDOWN*

Network is down. The operation encountered a dead network. This could indicate the failure of the network stack, the network interface, or the local network.

10051—*WSAENETUNREACH*

Network is unreachable. An operation was attempted to an unreachable network. This indicates that the local host does not know how to reach the remote host—in other words, no known route to the destination exists.

10052—*WSAENETRESET*

Network dropped the connection on reset. The connection has been broken because keepalives have detected a failure. This error can also occur when attempting to set the *SO_KEEPALIVE* option with *setsockopt* on a connection that has already failed.

10053—*WSAECONNABORTED*

Software caused the connection to abort. An established connection was aborted due to a software error. Typically, this means the connection was aborted due to a protocol or timeout error.

10054—*WSAECONNRESET*

Connection reset by peer. An established connection was forcibly closed by the remote host. This error can occur if the remote process is abnormally terminated (as in memory violation or hardware failure) or if a hard close was performed on the socket. A socket can be configured for a hard close using the *SO_LINGER* socket option and *setsockopt*. (For more information, see Chapter 9.)

10055—*WSAENOBUFS*

No buffer space available. The requested operation could not be performed because the system lacked sufficient buffer space.

10056—*WSAEISCONN*

Socket is already connected. A connection is being attempted on a socket that is already connected. This can occur on both datagram and stream sockets. When using datagram sockets, if *connect* or *WSAConnect* has been called to associate an endpoint's address for datagram communication, attempting to call either *sendto* or *WSASendTo* will generate this error.

10057—*WSAENOTCONN*

Socket is not connected. This error occurs when a request is made to send or receive data on a connection-oriented socket that is not currently connected.

10058—*WSAESHUTDOWN*

Cannot send after socket shutdown. The socket has already been partially closed by a call to *shutdown*, and either a send or a receive operation is being requested. Note that this occurs only on the data-flow direction that has been shut down. For example, after calling *shutdown* on sends, any call to send data will generate this error.

10060—*WSAETIMEDOUT*

Connection timed out. This error occurs when a connection request has been made and the remote computer fails to properly respond (or doesn't respond at all) after a specified length of time. This error is typically seen when the socket options *SO_SNDTIMEO* and *SO_RCVTIMEO* are set on a socket as well as when the *connect* and *WSAConnect* functions are called. For more information on setting *SO_SNDTIMEO* and *SO_RCVTIMEO* on a socket, see Chapter 9.

10061—*WSAECONNREFUSED*

Connection refused. The connection could not be established because the target machine refused it. This error usually occurs because no application on the remote machine is servicing connections on that address.

10064—*WSAEHOSTDOWN*

Host is down. This error indicates that the operation has failed because the destination host is down; however, an application is more likely to receive the error *WSAETIMEDOUT* because it typically occurs when attempting to establish a connection.

10065—*WSAEHOSTUNREACH*

No route to host. An operation was attempted to an unreachable host. This error is similar to *WSAENETUNREACH*.

10067—*WSAEPROCLIM*

Too many processes. Some Winsock service providers set a limit on the number of processes that can simultaneously access them.

10091—*WSASYSNOTREADY*

Network subsystem is unavailable. This error is returned when calling *WSAStartup*, and the provider cannot function because the underlying system that provides services is unavailable.

10092—*WSAVERNOTSUPPORTED*

Winsock.dll version out of range. The requested version of the Winsock provider is not supported.

10093—*WSANOTINITIALISED*

Winsock has not been initialized. A successful call to *WSAStartup* has not yet been performed.

10101—*WSAEDISCON*

Graceful shutdown in progress. This error is returned by *WSARecv* and *WSARecvFrom* to indicate that the remote party has initiated a graceful shutdown. This error occurs on message-oriented protocols such as ATM.

10102—*WSAENOMORE*

No more records found. This error is returned from *WSALookupServiceNext* to indicate that no additional records are left. This error is interchangeable with *WSA_E_NO_MORE*. Applications should check for both this error and *WSA_E_NO_MORE*.

10103—*WSAECANCELLED*

Operation canceled. This error indicates that a call to *WSALookupServiceEnd* was made while a call to *WSALookupServiceNext* was still processing. *WSALookupServiceNext* returns this error. This code is interchangeable with *WSA_E_CANCELLED*. Applications should check for both this error and *WSA_E_CANCELLED*.

10104—*WSAEINVALIDPROCTABLE*

The procedure call table is invalid. This error is typically returned by a service provider when the procedure table contains invalid entries. For more information on service providers, see Chapter 14.

10105—*WSAEINVALIDPROVIDER*

Invalid service provider. This error is associated with service providers and occurs when the provider cannot establish the correct Winsock version needed to function correctly.

10106—*WSAEPROVIDERFAILEDINIT*

The provider failed to initialize. This error is associated with service providers and is typically seen when the provider cannot load the necessary DLLs.

10107—*WSASYSCALLFAILURE*

System call failure. A system call that should never fail has failed.

10108—*WSASERVICE_NOT_FOUND*

No such service found. This error is normally associated with registration and name resolution functions when querying for services. (See Chapter 10 for more information about these functions.) This error indicates that the requested service could not be found in the given name space.

10109—*WSATYPE_NOT_FOUND*

Class type not found. This error is also associated with the registration and name resolution functions when manipulating service classes. When an instance of a service is registered, it must reference a service class that was previously installed with *WSAInstallServiceClass*.

10110—*WSA_E_NO_MORE*

No more records found. This error is returned from *WSALookupServiceNext* to indicate that no additional records are left. It is interchangeable with *WSAENOMORE*. Applications should check for both this error and *WSAENOMORE*.

10111—*WSA_E_CANCELLED*

Operation canceled. This error indicates that a call to *WSALookupServiceEnd* was made while a call to *WSALookupServiceNext* was still processing. *WSALookupServiceNext* returns this error. This code is interchangeable with *WSAECANCELLED*. Applications should check for both this error and *WSAECANCELLED*.

10112—*WSAEREFUSED*

Query refused. A database query failed because it was actively refused.

11001—*WSAHOST_NOT_FOUND*

Host not found. This error occurs with *gethostbyname* and *gethostbyaddr* to indicate that an authoritative answer host was not found.

11002—*WSATRY_AGAIN*

Nonauthoritative host not found. This error is also associated with *gethostbyname* and *gethostbyaddr*, and it indicates that either the nonauthoritative host was not found or a server failure occurred.

11003—*WSANO_RECOVERY*

A nonrecoverable error occurred. This error is also associated with *gethostbyname* and *gethostbyaddr*. It indicates that a nonrecoverable error has occurred and the operation should be tried again.

11004—*WSANO_DATA*

No data record of the requested type found. This error is also associated with *gethostbyname* and *gethostbyaddr*. It indicates that the supplied name was valid but that no data record of the requested type was found with it.

11005—*WSA_QOS_RECEIVERS*

At least one reserve message has arrived. This value is associated with IP Quality of Service (QOS) and is not an error per se. (See Chapter 12 for more on QOS.) It indicates that at least one process on the network is interested in receiving QOS traffic.

11006—*WSA_QOS_SENDERS*

At least one path message has arrived. This value is associated with QOS and is more of a status message. This value indicates that at least one process on the network is interested in sending QOS traffic.

11007—*WSA_QOS_NO_SENDERS*

No QOS senders. This value is associated with QOS and indicates that there are no longer any processes interested in sending QOS data. See Chapter 12 for a more complete description of when this error occurs.

11008—*WSA_QOS_NO_RECEIVERS*

No QOS receivers. This value is associated with QOS and indicates that there are no longer any processes interested in receiving QOS data. See Chapter 12 for a more complete description of this error.

11009—*WSA_QOS_REQUEST_CONFIRMED*

Reservation request has been confirmed. QOS applications can request that they be notified when their reservation request for network bandwidth has been approved. When such a request is made, this is the message generated. See Chapter 12 for a more complete description.

11010—*WSA_QOS_ADMISSION_FAILURE*

Error due to lack of resources. Insufficient resources were available to satisfy the QOS bandwidth request.

11011—*WSA_QOS_POLICY_FAILURE*

Invalid credentials. Either the user did not possess the correct privileges or the supplied credentials were invalid when making a QOS reservation request.

11012—*WSA_QOS_BAD_STYLE*

Unknown or conflicting style. QOS applications can establish different filter styles for a given session. This error indicates either unknown or conflicting style types. See Chapter 12 for a description of filter styles.

11013—*WSA_QOS_BAD_OBJECT*

Invalid *FILTERSPEC* structure or provider-specific object. This error occurs if either the *FLOWSPEC* structures or the provider-specific buffers of a QOS object are invalid. See Chapter 12 for more details.

11014—*WSA_QOS_TRAFFIC_CTRL_ERROR*

Problem with a *FLOWSPEC*. This error occurs if the traffic control component has a problem with the supplied *FLOWSPEC* parameters that are passed as a member of a QOS object.

11015—*WSA_QOS_GENERIC_ERROR*

General QOS error. This is a catchall error that is returned when the other QOS errors do not apply.

6—*WSA_INVALID_HANDLE*

Specified event object invalid. This Win32 error is seen when using Winsock functions that map to Win32 functions. This particular error occurs when a handle passed to *WSAWaitForMultipleEvents* is invalid.

8—*WSA_NOT_ENOUGH_MEMORY*

Insufficient memory available. This Win32 error indicates that insufficient memory is available to complete the operation.

87—*WSA_INVALID_PARAMETER*

One or more parameters are invalid. This Win32 error indicates that a parameter passed into the function is invalid. This error also occurs with *WSAWaitForMultipleEvents* when the event count parameter is not valid.

258—*WSA_WAIT_TIMEOUT*

Operation timed out. This Win32 error indicates that the overlapped operation did not complete in the specified time.

995—*WSA_OPERATION_ABORTED*

Overlapped operation aborted. This Win32 error indicates that an overlapped I/O operation was canceled because of the closure of a socket. In addition, this error can occur when executing the *SIO_FLUSH* ioctl command.

996—*WSA_IO_INCOMPLETE*

Overlapped I/O event object is not in a signaled state. This Win32 error is also associated with overlapped I/O. It is seen when calling *WSAGetOverlappedResults* and indicates that the overlapped I/O operation has not yet completed.

997—*WSA_IO_PENDING*

Overlapped operations will complete later. When making an overlapped I/O call with a Winsock function, this Win32 error is returned to indicate that the operation is pending and will complete later. See Chapter 8 for a discussion of overlapped I/O.

About the Authors

Anthony Jones

Anthony Jones was born in San Antonio, Texas, and graduated with honors from the University of Texas at San Antonio in 1996 with a bachelor's degree in computer science. His undergraduate thesis was based upon optimizing the Icon compiler.

After graduation, Anthony worked for Southwest Research Institute, a nonprofit contract research company in San Antonio. There he worked on a variety of projects, including real-time embedded control systems and visualization and simulation software, for customers ranging from the United States Air Force to The Weather Channel. In 1997, he moved to Washington to work for Microsoft Developer Support on the NetAPI team. Anthony recently moved to the Windows 2000 Core Networking department, where he is a tester on the Winsock team.

In his spare time, Anthony enjoys mountain biking, skiing, hiking, photography, and watching *Futurama* and *The X-Files*.

Jim Ohlund

Jim Ohlund works as a software design engineer for Microsoft's Proxy Server test team in Redmond, Washington. He has worked in many areas of the computer industry, from systems programming to developer support to software testing.

In 1990, Jim received a bachelor's degree in computer science from the University of Texas at San Antonio. Jim began his computer career while still in college by developing personnel systems for the United States Department of Defense. He expanded his working knowledge of computer networks and network programming in 1994 by developing terminal emulation software for Windows platforms. In 1996, Jim joined Microsoft's Developer Support Networking API team, helping software developers use many of the networking APIs described in this book.

When Jim is not working with computers, he likes to ski, bicycle, and hike in the beautiful Pacific Northwest.